Symbolische Ausführung eines RISC-V-Prozessors

Masterarbeit
Universität Bremen
Fachbereich 3

Jil Tietjen
<jiltietj@informatik.uni-bremen.de>

Erstgutachter: Prof. Dr. Rolf Drechsler
Zweitgutachter: Prof. Dr. Jan Peleska

Betreuung: Prof. Dr. Rolf Drechsler,
Dr. Daniel Große & Vladimir Herdt

Bremen, 12. September 2018

Universität Bremen

Nachname Tietjen Matrikelnr. 2812798

Vorname/n Jil

Diese Erklärungen sind in jedes Exemplar der Bachelor- bzw. Masterarbeit mit einzubinden.

Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Stellen, die ich wörtlich oder sinngemäß aus anderen Werken entnommen habe, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Datum Unterschrift

Erklärung zur Veröffentlichung von Abschlussarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten.
Archiviert werden:
1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils der ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.

☑ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit nach frühestens 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Datum Unterschrift

INHALTSVERZEICHNIS

1 EINLEITUNG

Schon seit einigen Jahrzehnten sind Rechner aus dem Leben der meisten Menschen nicht mehr wegzudenken. Das Kernstück eines Rechners und vieler anderer technischer Geräte ist der Prozessor. Er führt die zugrundeliegenden Operationen wie arithmetisch-logische Berechnungen oder Speicherzugriffe aus.

Der Befehlssatz eines Prozessors bestimmt die Menge an Befehlen, die im Prozessor ausgeführt werden können. Hierbei gibt es unterschiedliche Architekturen: Der *Reduced Instruction Set Computer* (RISC) ist eine Prozessorarchitektur mit einem stark vereinfachten Befehlssatz, der auf das Nötigste reduziert ist (siehe Asanović u. Patterson (2014)). Eine lizenzfreie und offene RISC-Befehlssatzarchitektur ist RISC-V, die viel Freiheit in der Software- und Hardware-Architektur bietet (siehe RISC-V Foundation (2015)). Diese wurde in Berkeley mit dem Ziel einer offenen Prozessorarchitektur für alle möglichen Einsatzzwecke entwickelt. RISC-V dient sowohl als Forschungsplattform als auch als Grundlage für kommerzielle Systeme, da es eine geringe Komplexität aufweist, leicht programmierbar und zukunftssicher entworfen ist. Es lassen sich schnelle und leistungsfähige Prozessoren mit wenig Chipfläche realisieren, die energieeffizient sind (siehe Oed u. Eckstein (2018)).

Im modernen Prozessorentwurf startet man zunächst mit einem abstrakten Simulationsmodell in Form eines *Instruction Set Simulators* (ISS). Das bedeutet, dass Programme simuliert werden können und sowohl die Softwareentwicklung selbst als auch die funktionale Überprüfung von Implementierungen erleichtert werden (siehe Herdt u. a. (2018)).

In der Arbeitsgruppe *Rechnerarchitektur* wurde im Rahmen der Vorlesung „Praktische Einführung in den modernen Systementwurf mit C++“ ein ISS sowie ein SystemC-basierter virtueller Prototyp des RISC-V entwickelt (siehe Herdt u. a. (2018)).

In der vorliegenden Arbeit wird ein Verfahren vorgestellt, welches es dem Softwareentwickler eines RISC-V-Prozessors erlaubt, Software-Assertions zu prüfen. Existierende Verfahren erlauben lediglich eine Überprüfung zur Laufzeit mit der aktuellen simulierten konkreten Eingabe. Hier wird ein neuartiger Ansatz vorgestellt, der auf Basis des oben genannten RISC-V-ISS eine symbolische Ausführung erlaubt.

Symbolische Ausführung ist eine effektive Technik, um herauszufinden, ob ein Programmstück gewisse Eigenschaften erfüllt oder nicht. Das Verfahren wird seit den 70er Jahren häufig als Programmanalysetechnik eingesetzt (siehe King (1976)). Bei einer konkreten Ausführung kann ein Programm nur mit einem konkreten Input ausgeführt werden und nur für diese Eingabe können Fehler in dem Programm entdeckt werden. Bestehende Fehler, die unabhängig von dieser Eingabe sind, werden dadurch nicht unbedingt gefunden. Bei der symbolischen Aus-

führung hingegen kann der Software-Entwickler definieren, welche Programmvariablen symbolisch behandelt werden sollen. Auf diesen symbolischen Variablen können Operationen ausgeführt werden, wobei die Ergebnisse als SMT-Ausdrücke gespeichert werden. Bei der Abarbeitung des Programmes werden mehrere Zustände traversiert. Das Besondere daran ist, dass bei einer if-Bedingung Zustände geclont werden und beide Zustände mit unterschiedlichen Werten ausgeführt werden können. So kann das Programm auf viele Fehler gleichzeitig getestet und die funktionale Korrektheit für die Menge der betrachteten Eingaben kann sichergestellt werden (siehe Cadar u. a. (2008)).

Der Aufbau der Arbeit setzt sich folgendermaßen zusammen: In Abschnitt 2 werden die nötigen Grundlagen zum Verständnis der Arbeit eingeführt. Die folgenden beiden Abschnitte bilden den Kern der Arbeit und beschreiben den Aufbau und Ablauf der symbolischen Ausführungs-Engine. Es wird mit einem Überblick über die symbolische Ausführung auf einem RISC-V-Prozessor begonnen. Als Erstes werden die symbolischen Register realisiert. Alle Instruktionen des RISC-V-Prozessors werden angepasst, sodass SMT-Ausdrücke verarbeitet werden können. Neben den Registern ist auch ein symbolischer Hauptspeicher notwendig. Sobald ein Branching auftritt, müssen Zustände geclont werden. Somit gibt es eine Exploration des Zustandsraumes und es können Strategien angewendet werden, welcher Zustand als nächstes gewählt werden soll. In Abschnitt 5 folgt die Evaluation des RISC-V-Prozessors. Hierfür werden unterschiedliche Eingabeprogramme in C verwendet, die wiederum verschiedene Assumptions und Assertions aufweisen. Es kann herausgefunden werden, ob alle eingebauten Fehler im Programm mit der symbolischen Ausführungs-Engine gefunden werden können. Des Weiteren werden die Eingabeprogramme anhand von zwei Explorationsstrategien untersucht. Es wird einmal durch die Tiefensuche der nächste abzuarbeitende Zustand ausgewählt oder der nächste Zustand wird rein zufällig bestimmt. Zum Schluss werden alle Kernpunkte der Arbeit zusammengefasst und ein Ausblick für mögliche Erweiterungen gegeben.

2 GRUNDLAGEN

In diesem Kapitel werden alle nötigen Grundlagen für das Verständnis dieser Masterarbeit beschrieben. Es dient als Einführung in das Thema der RISC-V-ISA und zum Verständnis der symbolischen Ausführung.

2.1 *RISC-V-ISA*

Ein RISC-V-Prozessor implementiert eine ISA, die auf einer RISC-Architektur basiert. *Reduced Instruction Set Computer* (RISC) steht für einen Computer mit einem reduzierten Befehlssatz und hat eine kleine Menge an einfachen und allgemeinen Anweisungen. Es gibt nur wenige Adressierungsarten und nur load/store-Befehle, die auf den Speicher direkt zugreifen. Als Speichersystem verwendet die ISA Little Endian. Daher wird das am wenigsten signifikante Bit an der niedrigsten Speicheradresse gespeichert.

RISC-V ist eine ISA (Instruction Set Architecture), entwickelt von der RISC-V-Foundation (siehe RISC-V Foundation (2015)), die frei zugänglich ist und für die jeweiligen Einsatzgebiete angepasst werden kann. In dieser Arbeit wird der ISS der Vorlesung „Praktische Einführung in den modernen Systementwurf mit C++“ (siehe Abschnitt 2.2) verwendet. Eingesetzt wird die Standard Erweiterung RV32IMA, die die Basisintegeroperationen zusammen mit Mul-/Div-Instruktionen und atomaren Operationen enthält. Insgesamt hat die in dieser Arbeit verwendete RISC-V-ISA 45 Instruktionen. Diese 45 Instruktionen gliedern sich in 5 Klassen auf: Branching, load/store und arithmetische- sowie logische Befehle, *ECALL* und *EBREAK*. Diese unterschiedlichen Instruktionen werden durch sechs Instruktionsformate (R/I/S/B/U/J) umgesetzt, wie in Abbildung 1 zu sehen.

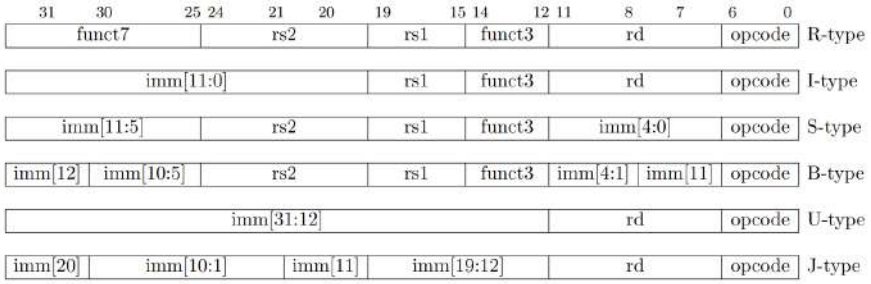


Abbildung 1: Instruktionsformate (siehe RISC-V Foundation (2015))

Jede Instruktion ist 32 Bit lang und alle Instruktionen enthalten den Opcode an der Bitstelle $0-6$ (rechte Seiten in Abbildung 1). Der Opcode gibt an, um welchen Maschinenbefehl es sich handelt. Zum Beispiel ist *ADD* ($b0110011$) der Opcode für eine Addition. Die verbleibenden 25 Bits werden abhängig vom Instruktionsformat unterschiedlich aufgeteilt.

Der *R-type* hat das Zielregister (r_d) an der Bitstelle $7 - 11$. In dem Zielregister wird das Ergebnis einer Instruktion gespeichert. Die beiden Sourceregister r_{s1} und r_{s2} sind an den Bitstellen $15 - 19$ und $20 - 24$ zu finden. Der Bereich von *funct3* ist ein zusätzliches Opcode-Feld (Bitstelle $12 - 14$), sowie auch *funct7* von Bitstelle 25 bis 31.

Beim *I-type* gibt es anstelle von *funct7* und r_{s2} eine Konstante (Immediate), die 12 Bit breit ist. Es kann offensichtlich nur eine begrenzte Größe an Zahlen im Immediate enthalten sein. Dabei wird zwischen signed und unsigned unterschieden. Bei unsigned können alle Werte zwischen 0 und 4095 dargestellt werden und bei signed zwischen -2048 und 2047.

Beim *U-type* gibt es außer dem Opcode und dem Zielregister ein 20 Bit breites Immediate. Es wird auf 32 Bit erweitert, indem die niederwertigen Bits mit Nullen gefüllt werden. Das heißt, es wird um 12 Bit nach links geshiftet.

Der *J-type* ist identisch zum *U-type*. Allerdings wird das Immediate nur um ein Bit geshiftet, um Jump-Adressen zu kodieren.

Der *S-type* enthält neben dem Opcode, zwei Sourceregister, *funct3*, ein Immediate von 4 Bit anstelle des Zielregisters und ein Immediate von 6 Bit.

Im Gegensatz zum *S-type* kodiert der *B-type* den Offset vom Sprung in einem unterteilten 12 Bit Immediate.

Tabelle 1 zeigt alle Instruktionen der RISC-V-ISA mit den entsprechenden Formaten. Zusätzlich wird der Befehl in Assembler angegeben (Spalte *Syntax*) und seine Bedeutung erklärt (Spalte *Operation*). In der Tabelle stehen *U* für unsigned und *S* für signed über den Operationen, sodass es zum Beispiel eine Multipliaktion von einem signed Wert und einem unsigned Wert geben kann (*MULHSU*). Bei Shift-Operationen wird $\ll$ für arithmetische und $\lll$ für logische Operationen als links-shift benutzt, äquivalente Anwendung für $\gg$ als rechts-shift. Bei bitweisen Operationen werden die logischen Operationen & für *AND*, | für *OR* oder $\wedge$ für *XOR* verwendet. Die Abkürzung *pcnt* steht für den program counter. Dieser zeigt immer auf die nächste auszuführende Instruktion.

In der Tabelle 1 wird bei einigen Instruktionen *imm* oder *shamt* verwendet. Ein Immediate kann eine Länge von 12 beziehungsweise 20 Bit haben. 20-Bit-Werte sind hierbei vorzeichenbehaftet und 12-Bit-Werte sind grundsätzlich positiv. Sie werden immer vor der Verwendung auf die Breite des Registers erweitert. Bei der Load-Instruktion wird zum Beispiel die effektive Byteadresse zu dem 12-Bit-Offset (*imm*) hinzugefügt.

Anstatt des Immediates wird bei Shift-Operationen ein shift amount (*shamt*) verwendet. Dabei gibt es keinen Unterschied im Verhalten zum Immediate.

Die Instruktionen *ECALL* und *EBREAK* haben unterschiedliche Effekte, abhängig davon, in welcher Umgebung sie verwendet werden.

Tabelle 1: Instruktionen im RISC-V-ISA (Teil 1)

Name	Operation	Syntax	Format
LUI	$\$r_d$:= (imm $\ll$ 12)	lui $\$r_d$, imm	U-type
AUIPC	$\$r_d$:= pcnt + (imm $\ll$ 12)	auipc $\$r_d$, imm	U-type
JAL	$\$r_d$:= pcnt + 4; pcnt = pcnt + imm	jal label	J-type
JALR	$\$r_d$ = pcnt + 4; pcnt = $\$r_{s1}$ + imm	jalr $\$r_d$, $\$r_{s1}$, label	I-type
BEQ	if $\$r_{s1}$ = $\$r_{s2}$ then pcnt = pcnt + imm	beq $\$r_{s1}$, $\$r_{s2}$, label	B-type
BNE	if $\$r_{s1}$ $\neq$ $\$r_{s2}$ then pcnt = pcnt + imm	bne $\$r_{s1}$, $\$r_{s2}$, label	B-type
BLT	if $\$r_{s1}$ < $\$r_{s2}$ then pcnt = pcnt + imm	blt $\$r_{s1}$, $\$r_{s2}$, label	B-type
BGE	if $\$r_{s1}$ $\geq$ $\$r_{s1}$ then pcnt = pcnt + imm	bge $\$r_{s1}$, $\$r_{s2}$, label	B-type
BLTU	if $\$r_{s1}$ $\overset{U}{<}$ $\$r_{s2}$ then pcnt = pcnt + imm	bltu $\$r_{s1}$, $\$r_{s2}$, label	B-type
BGEU	if $\$r_{s1}$ $\overset{U}{\geq}$ $\$r_{s1}$ then pcnt = pcnt + imm	bgeu $\$r_{s1}$, $\$r_{s2}$, label	B-type
LB	$\$r_d$ = mem[$\$r_{s1}$ + imm]	lb $\$r_d$, imm($\$r_{s1}$)	I-type
LH	$\$r_d$ = mem[$\$r_{s1}$ + imm...$\$r_{s1}$ + imm+1]	lh $\$r_d$, imm($\$r_{s1}$)	I-type
LW	$\$r_d$ = mem[$\$r_{s1}$ + imm...$\$r_{s1}$ + imm+3]	lw $\$r_d$, imm($\$r_{s1}$)	I-type
LBU	$\$r_d$ = mem[$\$r_{s1}$ + imm] & #xFF	lbu $\$r_d$, imm($\$r_{s1}$)	I-type
LHU	$\$r_d$ = mem[$\$r_{s1}$ + imm...$\$r_{s1}$ + imm+1] & #xFFFF	lhu $\$r_d$, imm($\$r_{s1}$)	I-type
SB	mem[$\$r_{s1}$ + imm] = $\$r_{s2}$ & #xFF	sb $\$r_{s2}$, imm($\$r_{s1}$)	S-type
SH	mem[$\$r_{s1}$ + imm ... $\$r_{s1}$ + imm + 1] = $\$r_{s2}$ & #xFFFF	sh $\$r_{s2}$, imm($\$r_{s1}$)	S-type
SW	mem[$\$r_{s1}$ + imm ... $\$r_{s1}$ + imm + 3] = $\$r_{s2}$	sw $\$r_{s2}$, imm($\$r_{s1}$)	S-type
ADDI	$\$r_d$ = $\$r_{s1}$ + imm	addi $\$r_d$, $\$r_{s1}$, imm	I-type
SLTI	if $\$r_{s1}$ < imm then $\$r_d$ = 1 else $\$r_d$ = 0	slti $\$r_d$, $\$r_{s1}$, imm	I-type

Übliche Umgebungen sind Betriebssysteme, wo *ECALL* für Systemcalls verwendet wird. Der Befehl *EBREAK* wird oft in Debug-Programmen eingesetzt, um die Steuerung zurück zu einem Debug-Programm zu übertragen. In dieser Arbeit wird nur der Befehl *ECALL* verwendet, um die Assumptions und Assertions in der symbolischen Ausführungs-Engine auszudrücken.

Zum Schluss sind noch einige Besonderheiten bei den Instruktionen *Div* und *Rem* bei der Division durch 0 oder Modulo mit 0 anzumerken (siehe Tabelle 2). Bei der Modulo-Berechnung mit 0 wird der Dividend, also der erste Operand, als Ergebnis festgelegt. Bei der Division ist bei *DIVU* der höchstmögliche Wert das Ergebnis und bei *DIV* ist das Ergebnis immer -1.

Vorzeichenbehafteter Divisionsüberlauf tritt nur auf, wenn die kleinstmögliche negative Ganzzahl durch -1 geteilt wird. Das Ergebnis ist dann der Divident.

Tabelle 1: Instruktionen im RISC-V-ISA (Teil 2)

Name	Operation	Syntax	Format
SLTIU	if $\$r_{s1} \overset{U}{<}$ imm then $\$r_d = 1$ else $\$r_d = 0$	sltiu $\$r_d$, $\$r_{s1}$, imm	I-type
XORI	$\$r_d = \$r_{s1} \hat{}$ imm	xori $\$r_d$, $\$r_{s1}$, imm	I-type
ORI	$\$r_d = \$r_{s1} \mid$ imm	ori $\$r_d$, $\$r_{s1}$, imm	I-type
ANDI	$\$r_d = \$r_{s1}$ & imm	andi $\$r_d$, $\$r_{s1}$, imm	I-type
SLLI	$\$r_d = \$r_{s1} \ll$ shamt	slli $\$r_d$, $\$r_{s1}$, shamt	I-type
SRAI	$\$r_d = \$r_{s1} \gg$ shamt	srai $\$r_d$, $\$r_{s1}$, shamt	I-type
ADD	$\$r_d = \$r_{s1} + \$r_{s2}$	add $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
SUB	$\$r_d = \$r_{s1} - \$r_{s2}$	sub $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
SLL	$\$r_d = \$r_{s1} \lll \$r_{s2}$	sll $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
SLT	if $\$r_{s1} < \$r_{s2}$ then $\$r_d = 1$ else $\$r_d = 0$	slt $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
SLTU	if $\$r_{s1} \overset{U}{<} \$r_{s2}$ then $\$r_d = 1$ else $\$r_d = 0$	sltu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
XOR	$\$r_d = \$r_{s1} \hat{} \$r_{s2}$	xor $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
SRL	$\$r_d = \$r_{s1} \ggg \$r_{s2}$	srl $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
SRA	$\$r_d = \$r_{s1} \gg \$r_{s2}$	sra $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
OR	$\$r_d = \$r_{s1} \mid \$r_{s2}$	or $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
AND	$\$r_d = \$r_{s1}$ & $\$r_{s2}$	and $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
ECALL	umgebungsabhängig	ecall	I-type
EBREAK	umgebungsabhängig	ebreak	I-type
MUL	$\$r_d = \$r_{s1} * \$r_{s2}$	mul $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
MULH	$\$r_d = (\$r_{s1} * \$r_{s2}) \gg 32$	mulh $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
MULHSU	$\$r_d = (\$r_{s1} \overset{SU}{*} \$r_{s2}) \gg 32$	mulhsu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
MULHU	$\$r_d = (\$r_{s1} \overset{UU}{*} \$r_{s2}) \gg 32$	mulhsu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
DIV	$\$r_d = (\$r_{s1} / \$r_{s2})$	div $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
DIVU	$\$r_d = \$r_{s1} \overset{U}{/} \$r_{s2}$	divu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
REM	$\$r_d = \$r_{s1} \% \$r_{s2}$	rem $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type
REMU	$\$r_d = \$r_{s1} \overset{U}{\%} \$r_{s2}$	remu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	R-type

Tabelle 2: Semantik von Division und Modulo (siehe Waterman u. Asanovìc (2017))

Bedingung	Divident	Divisor	DIVU	REMU	DIV	REM
Division durch 0	x	0	$2^{32}-1$	x	-1	x
Overflow (signed)	-2^{32-1}	-1	-	-	-2^{32-1}	0

2.2 *RISC-V-ISS*

Der von der Rechnerarchitektur zur Verfügung gestellte ISS wurde in C++ implementiert und der Grundaufbau ist vereinfacht in Abbildung 2 dargestellt.

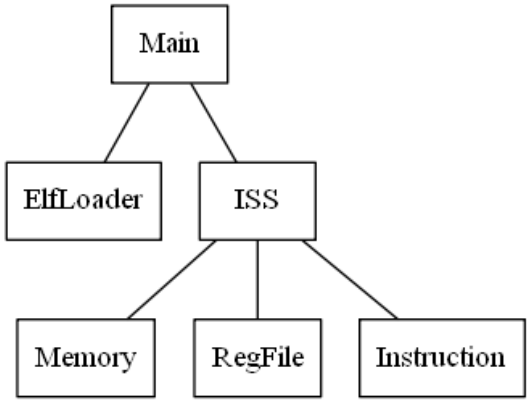


Abbildung 2: Vereinfachtes Klassendiagramm des RISC-V-ISS

In der Klasse *ISS* sind alle Instruktionen des RISC-V-Prozessors implementiert. Die Register werden initialisiert in der Klasse *RegFile*. In *Memory* wird der Hauptspeicher verwaltet. Die Klasse *Instruction* kümmert sich um die Behandlung und Dekodierung von Instruktionen. Der *ElfLoader* lädt die Executables, die simuliert werden sollen.

2.3 *Symbolische Ausführung*

Symbolische Ausführung ist ein mächtiges Verfahren in der Softwareentwicklung, um Programme zu verifizieren. Mit Hilfe von symbolischer Ausführung können Programme mit symbolischen Werten ausgeführt werden. Dadurch wird anstelle eines konkreten Programmpfades eine Menge von Pfaden betrachtet und es können verschiedene Bedingungen durch symbolische Variablen im Programm definiert werden. Auf diese Weise können Bedingungen allgemein geprüft werden und man erhält ein Verfahren zur Verifikation von Programmen.

Zur Umsetzung der symbolischen Ausführung werden symbolische Werte eingeführt, um die Werte von Programmvariablen als symbolische Ausdrücke darzustellen. Im nächsten Schritt werden die berechneten Ausgabewerte eines Programms ebenfalls durch eine Funktion der eingegebenen symbolischen Werte ausgedrückt (siehe Cadar u. Koushik (2013)). Es gibt daher Mengen von Eingaben, beschränkt durch Annahmen (Assumptions), und Mengen von Werten für Variablen. Diese Mengen werden jeweils durch SMT-Constraints (ausführlicher in Abschnitt 2.4) definiert.

Zu jedem Pfad im Programm wird eine SMT-Formel über die symbolischen Eingaben erstellt, welche die Pfadbedingung darstellt. Die SMT-Constraints der Pfadbedingung, der program counter und der Speicher bilden einen Zustand. Die Pfadbedingung definiert also Constraints, die die Eingaben erfüllen müssen, damit die Ausführung

einem bestimmten Pfad in dem Programm folgt. Der program counter definiert immer die nächste auszuführende Instruktion im Programm.

Ein Ausführungsbaum (Execution Tree) kennzeichnet die zu verfolgenden Pfade innerhalb der Ausführung. Dabei sind die Knoten die Programmzustände und die Kanten stellen die Übergänge zwischen den Zuständen dar.

Zu Beginn der Ausführung eines Programmes ist die Pfadbedingung *true*. Bei jeder weiteren Assumption wird die Pfadbedingung durch entsprechende SMT-Ausdrücke ergänzt. Um herauszufinden, ob ein Programm fehlerfrei ist, werden durch den Benutzer spezifizierte Assertions hinzugezogen. Ein SMT-Solver überprüft, ob eine Assertion erfüllt wird oder nicht. Ist die negierte Assertion anhand des symbolischen Eingabewertes erfüllbar, gibt der SMT-Solver SAT zurück.

Der Ablauf wird im Folgenden anhand eines Beispiels verdeutlicht:

Listing 1: Eingabeprogramm

```
1  int main(){
2    make_symbolic(a);
3    int a, b = 5, c = 0;
4    assume(a < 12 && a > 2);
5    while(a < 10){
6      c++;
7      if(b > a){
8        a += c;
9        b -= 4;
10     }else{
11       b *= 2;
12     }
13   }
14   assert (c > 6);
15   return 0;
16 }
```

Jede Programmzeile im Eingabeprogramm stellt mindestens einen Zustand im Execution Tree dar. Zusätzlich enthält jeder Zustand die Pfadbedingung π und die Abbildung der Variablen des Eingabeprogramms auf die entsprechenden Werte durch σ. In diesem Programm ist der Integer *a* symbolisch. Bei einer Assertion wird der aktuelle Zustand gegen diese evaluiert. Der Ausführungsbaum ist in Abbildung 3 zu sehen.

Man beginnt mit Zustand *A* und initialisiert erst einmal alle Variablen und die Pfadbedingung ist zu Beginn *true*. Zustand B enthält die Assumption aus Zeile 4 aus Listing 1, daher wird die Pfadbedingung durch neue SMT-Ausdrücke angepasst. In Codeziele 5 beginnt die while-Schleife. Für die Pfadbedingung und für die Werte der Variablen erfolgen hier noch keine Änderungen. Allerdings führt die Schleife dazu, dass sich die Zustände verzweigen. Die Zustände werden auf

ihre Erreichbarkeit geprüft und sobald sich eine Pfadbedingung als unerfüllbar erweist, wird dieser Zustand verworfen. Abhängig von der genommenen Verzweigung wird als nächstes ein anderer Zustand ausgewertet und es werden unterschiedliche Annahmen für die symbolische Variable *a* getroffen. Es gibt nun einen neuen Zustand, bei dem die Schleifenbedingung erfüllt ist (*D*) und einen Zustand, bei dem sie nicht erfüllt ist (*K*). Dies wird in der Pfadbedingung definiert, indem die Schleifenbedingung an die Pfadbedingung angehängt wird. Dies geschieht ebenfalls einmal negiert für den Zustand, der aus der Schleife herausspringt (Zustand *K*). Zustand *K* wird somit direkt zur Assertion überführt und diese wird zu UNSAT evaluiert.

Zustand *D* erhöht als nächstes die Variable c und der neue Wert wird in der entsprechenden Variable abgespeichert. In Codezeile 7 gibt es eine if-Bedingung, die ebenfalls wieder zu einem Branching führt nach dem gleichen Prinzip wie bei der while-Schleife. Die Pfadbedingungen sind entsprechend der Zustände angepasst und es entstehen daraus die Zustände *G* und *F*. Das Programm wird weiter abgearbeitet und erreicht jeweils in Zustand *H* und *J* wieder die while-Schleife. Das Branchen wiederholt sich so lange, bis alle Pfade enden oder die Assertion fehlschlägt.

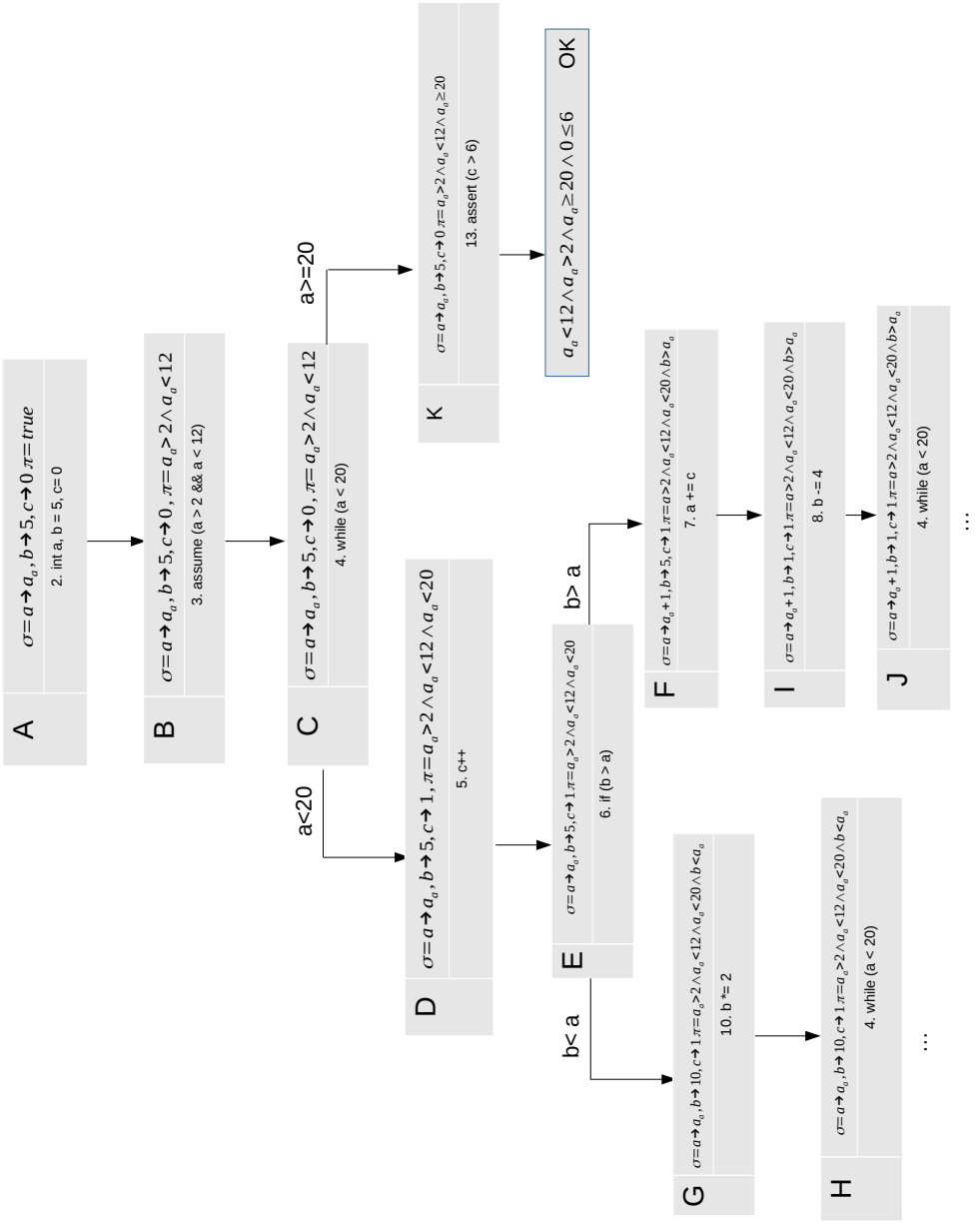


Abbildung 3: Ausführungsbaum

2.4 *SMT-Beweiser*

Bei symbolischer Ausführung wird ein SMT-Beweiser benötigt, um festzustellen, ob ein SMT-Ausdruck erfüllbar ist oder nicht. SMT-Solver können allgemein sehr komplexe Probleme lösen und werden eingesetzt zur Verifikation, Analyse und zum Testen von Programmen (siehe Bjørner u. de Moura (2014)).

SMT ist eine Erweiterung von SAT. SAT ist das Erfüllbarkeitsproblem in der Aussagenlogik. Das bedeutet, dass der SAT-Solver eine aussagenlogische Formel als Eingabe erhält, bei der herausgefunden werden soll, ob sie erfüllbar ist oder nicht. Dafür wird nach einer erfüllenden Belegung der Variablen in der Formel gesucht, so dass die gesamte Formel zu wahr auswertet. SAT ist NP-vollständig und daher vermutlich nicht in polynomieller Zeit lösbar.

In *Satisfiability Modulo Theories* (SMT) wird die Aussagenlogik ergänzt durch zusätzliche Theorien, wie beispielsweise Bitvektorarithmetik oder Linear Arithmetic (siehe Kroening u. Strichman (2008)).

In dieser Arbeit werden nur die Bitvektoren verwendet, die aus einer Folge von Bits bestehen. Ein Bitvektor ist ein Element $b = (b_{n-1}, \ldots, b_0) \in \mathbb{B}^n$. Die Funktion $[] : \mathbb{B}^n \times [0, n) \to \mathbb{B}$ bildet einen Bitvektor b und einen Index i auf das ite Element des Vektors ab (zum Beispiel $b[i] = b_i$). Operationen auf Bitvektoren fester Größe sind zum Beispiel *bvadd, bvsub* oder *bvashr* (siehe Kroening u. Strichman (2008)). Die Bitvektoren modellieren die bitbasierte Arithmetik von Rechnern genauer als ganzzahlige Logik. Zusätzlich sind hier bitweise Logikoperationen und Shifts möglich.

Für die SMT-Ausdrücke gibt es spezielle Solver, wie zum Beispiel MathSAT, OpenSMT oder Z3 (siehe Bjørner u. de Moura (2008)). In dieser Arbeit wird der Z3-Solver als SMT-Solver verwendet. Die Bitvektoren werden als eine Menge von Booleschen Variablen dargestellt und durch Bitvektoroperationen kodiert. Die Ausdrücke werden dann durch einen SMT-Solver gelöst (siehe Limaye u. Seshia (2010)). Der SMT-Solver gibt für die Erfüllbarkeit einer Formel SAT oder UNSAT zurück. Bei SAT liefert er außerdem eine konkrete Belegung zurück, um die Formel zu erfüllen. Bei UNSAT kann die Formel nicht erfüllt werden.

SMT-Solver basieren auf dem DPPL(T) Algorithmus. Das bedeutet, dass es eine Kommunikation zwischen DPLL (Algorithmus von gängigen SAT-Solvern) und dem T-Solver geben muss. Durch DPLL nicht definierbare Teilzuordnungen werden an den Theorie-Solver übergeben, der eine Entscheidung vornehmen soll. Gibt es einen Konflikt bei der Belegung, dann muss der T-Solver eine Erklärung für den Konflikt liefern, um mit der DPLL-Suche zu interagieren (siehe Dutertre u. de Moura (2006)).

3 GRUNDAUFBAU DER SYMBOLISCHEN AUSFÜHRUNG

In diesem Abschnitt wird die grundlegende symbolische Ausführung beschrieben. Dies geschieht in einzelnen Schritten, die aufeinander aufbauen. Zuerst wird im nächsten Abschnitt ein Überblick über die symbolische Ausführungs-Engine gegeben und ihre Architektur vorgestellt.

Basierend darauf wird dann vorgestellt, wie durch den Software-Entwickler symbolische Variablen, Assumptions und Assertions definiert werden können (siehe Abschnitt 3.2) und wie symbolische Eingaben verarbeitet werden (siehe Abschnitt 3.3). Daraufhin wird in Abschnitt 3.4 beschrieben, wie die symbolischen Variablen dazu führen, dass die Instruktionen der symbolischen Ausführungs-Engine angepasst werden.

Zunächst werden Assemblerprogramme als Eingabeprogramme betrachtet. Im weiteren Verlauf werden dann Programme auf C-Ebene in Abschnitt 3.5 und 3.6 eingeführt.

3.1 *Überblick und Architektur*

Im Gegensatz zum ISS kann die symbolische Ausführungs-Engine ein Programm nicht mehr nur konkret ausführen, sondern auch symbolisch.

Dafür werden Assumptions und Assertions im Assembler-Code definiert. Diese werden dann auf Ebene der symbolische Ausführungs-Engine umgesetzt.

Bei der symbolischen Abarbeitung eines Programmes traversiert der Sym-ISS mehrere symbolische Zustände. Ein Zustand ist ein Tupel $(\sigma, \pi, pcnt)$. σ ist der symbolische Speicher, der Register (Abschnitt 3.2) und Zellen des Hauptspeichers (Abschnitt 3.5) auf symbolische Werte abbildet. π ist die Pfadbedingung als SMT-Ausdruck. Die Pfadbedingung definiert die Bedingungen für die symbolischen Variablen und der SMT-Solver überprüft, ob der Zustand erreichbar ist. Wenn der SMT-Solver UNSAT zurück gibt, dann kann der Zustand nicht erreicht werden und wird entfernt.

Bei der Abarbeitung eines Programmes können unterschiedliche Ausführungspfade entstehen. Im Falle einer if-Bedingung wird diese einmal positiv und einmal negiert an die Pfadbedingung angehängt. Beide Bedingungen werden mit Hilfe des SMT-Solvers überprüft. Falls nur eine erfüllbar ist, wird mit der entsprechenden Verzweigung fortgefahren. Sind beide Bedingungen erfüllbar, muss der Zustand kopiert (geclont) werden. So entstehen für den then-Teil und den else-Teil der if-Bedingung jeweils ein Zustand.

Damit Berechnungen wie eine Addition oder Subtraktion auf symbolischen Registern ausführbar sind, werden die Instruktionen der RISC-V-ISA angepasst (siehe Abschnitt 3.4).

Im Folgenden wird einmal anhand eines Beispiels veranschaulicht, wie die Zustände entstehen. Es gibt ein symbolisches Register $a0$, das durch die Assumption $0 \leq a0 \leq 10$ eingeschränkt ist. Die aktuelle Pfadbedingung sei π_0. Das Programm berechnet die Summe von Register $a0$ und dem Immediate 12. Das Ergebnis der Addition kann zwischen 12 und 22 liegen. Wird als Nächstes ein Branchbefehl mit der Bedingung $(a0 < 18)$ ausgeführt, so entstehen zwei Zustände. Der eine Zustand enthält die Pfadbedingung, $\pi_0 \wedge (a0 < 18)$ und der andere Zustand die Pfadbedingung $\pi_0 \wedge (a0 \geq 18)$. Die symbolischen Bedingungen können durch den SMT-Solver dann zu true oder false ausgewertet werden (ausführliche Erklärung siehe Abschnitt 4). Je nachdem welchen symbolischen Wert $a0$ enthält, ist nur einer der beiden Zustände möglich. Der andere Zustand wird dann nicht mehr weiter betrachtet.

Im Folgenden wird die Architektur der symbolischen Ausführungs-Engine beschrieben (siehe Abbildung 4). Es wird gezeigt, wie die Klassen aufgebaut sind, um ein besseres Verständnis für die Umsetzung der Engine zu erhalten.

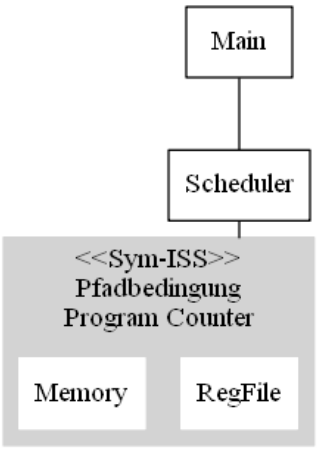


Abbildung 4: Vereinfachtes Klassendiagramm der symbolischen Ausführungs-Engine

In der Klasse *Sym-ISS* sind alle Instruktionen des RISC-V-ISA implementiert. In der Klasse wird der *Memory* geladen und die Instruktionen aus dem Programmcode werden nacheinander abgearbeitet. Die Register werden in der Klasse *RegFile* initialisiert und als symbolisch deklariert. Hierbei hält jedes der Register einen SMT-Ausdruck. In *Memory* wird der Hauptspeicher verwaltet. Hier gibt es Methoden zum Laden und Speichern von Bytes oder ganzen Datenwörtern. Durch

den *Scheduler* wird anhand von verschiedenen Explorationsstrategien der nächste Zustand des Sym-ISS ausgewählt.

In den nächsten Abschnitten wird anhand des Top-Down-Prinzips der Vorgang der symbolischen Ausführung eines RISC-V-Prozessors erklärt. Begonnen wird mit einem Programm bei dem symbolische Register, Assumptions und Assertions verwendet werden.

3.2 *Symbolische Register mit Assertions und Assumptions*

Um eine symbolische Ausführung nun durchführen zu können, werden einzelne Register als symbolisch definiert. Alle Register sind mit 0 initialisiert. Zusätzlich definiert der Software-Entwickler Assumptions und Assertions. Zur Umsetzung im RISC-V-ISA wird die Instruktion *ECALL* verwendet. Hierbei steht *ECALL* für *Environment Call* und ermöglicht das Senden einer Anfrage an eine Ausführungsumgebung. Die Ausführungsumgebung ist in dieser Arbeit die symbolische Ausführungs-Engine.

Der *ECALL* hat einen zusätzlichen Parameter, mit dem man übergeben kann, ob es sich um ein symbolisches Register, eine Assumption oder eine Assertion handelt. Der Parameter wird im Register $t0$ an den Prozess übergeben. Im Folgenden wird die Instrukiton *ECALL* einmal aufgelistet:

$$
\begin{aligned}
&li\ t0\ 3/2/1\\
&li\ t1\ REG\_NUMBER\\
&ECALL
\end{aligned}
$$

Der Wert in Register $t0$ legt fest, ob es sich um ein symbolisches Register, eine Assumption oder um eine Assertion handeln soll. Bei einer 1 handelt es sich um eine Assertion, bei einer 2 um eine Assumption und bei einer 3 um ein symbolisches Register.

Tritt Fall 3) ein, so wird in Register $t1$ die Registernummer des symbolischen Registers geladen. Zum Beispiel hat das Register $a0$ die Registernummer 10.

Im Fall 2) wird in Register $t1$ die Registernummer desjenigen Registers gespeichert, das die Assumption enthält. Enthält $t1$ beispielsweise den Wert 10, dann befindet sich die Assumption in Register $a0$.

Bei 1) wird analog zu Fall 2) vorgegangen mit dem Unterschied, dass das entprechende Register nun die Assertion statt der Assumption enthält.

Die Instruktion *ECALL* wird in der symbolischen Ausführungs-Engine gesondert behandelt, um Assumptions, Assertions und symbolischen Register auszuführen (siehe Abschnitt 3.3).

Zur Veranschaulichung wird im folgenden ein konkretes Beispielprogramm vereinfacht aufgeführt, bei dem eine Addition von einem Register mit dem konstanten Wert 10 und der Zahl 12 berechnet wird.

```
1 #a0:=10
2 li a0,10
3
4 #a1:=a0+12
5 addi a1,a0,12
```

Das gleiche Beispielprogramm wird nun durch ein symbolisches Register $a0$, um eine Assumption und um eine Assertion ergänzt.

```
1 #symbolisches Register = a0
2 li t1, 10 #RegNumber 10 = a0
3 li t0, 3
4 ecall
5
6 #Assumption Bedingung, gespeichert in t2 (Zeile 12)
7 #t2:=a0 < 13
8 slti t2,a0,13
9 #a4:=0 < a0
10 slt a4,zero,a0
11 #t2:=t2 ∧ a4 = 0 < a0 < 13
12 and t2,t2,a4
13
14 #Assumptionaufruf
15 #RegNumber 7 = t2
16 li t1, 7
17 li t0,2
18 ecall
19
20
21 #a1:=a0+12
22 addi a1,a0,12
23
24 #Assertion Bedingung
25 #t2:=a1<24
26 slti t2,a1,24
27
28 #Assertionaufruf
29 #RegNumber 7 = t2
30 li t1, 7
31 li t0, 1
32 ecall
```

Die Bedingung der Assumption setzt sich aus den Codezeilen 8 bis 12 zusammen. Da die Bedingung der Assumption in Register $t2$ zur weiteren Verarbeitung gespeichert werden soll, gibt es in $t1$ einen Verweis zu Register $t2$ (Codezeile 16). Die Assumption für dieses Programm ist, dass $a0$ einen Wert zwischen 1 und 12 annimmt ($0 < a0 < 13$).

In Zeile 22 findet die gleiche Berechnung statt wie in dem konkreten Beispielprogramm. Allerdings ist für $a0$ jeder Wert aus dem eben genannten Intervall möglich.

In Zeile 26 wird schlussendlich noch die Bedingung der Assertion definiert. Sobald $a1$ einen größeren Wert annimmt als 23, schlägt die Assertion fehl. In Zeile 32 geschieht dann die Ausführung im Sym-ISS.

In diesem Abschnitt wurde die Benutzersicht beschrieben, wie dieser die Eingabeprogramme verändern muss, damit eine symbolische Ausführung auf der Ausführungs-Engine möglich ist.

3.3 *Umsetzung der symbolischen Ausführungs-Engine*

Hier wird die technische Umsetzung in der Klasse *Sym-ISS* beschrieben. Dazu gehört, wie die Instruktion *ECALL* implementiert wird, damit der SMT-Solver die entsprechenden Pfadbedingungen auswerten kann und wie die Inhalte der Register verarbeitet werden.

Zu Beginn der Ausführung wird ein Zustand mit 32 Registern erstellt, die einen *SMT-Bitvektor* der Länge 32 enthalten. Alle Register werden mit 0 initialisiert. Die Pfadbedingung wird am Anfang auf true gesetzt.

In Listing 2 wird der Fall behandelt, dass ein Register symbolisch gemacht werden soll. Zu Beginn werden die Werte der Register $t0$ und $t1$ abgefragt (Codezeile 3, 6). Es wird davon ausgegangen, dass diese konstante Werte enthalten. Bei einem symbolischen Register würde $t0$ den Wert 3 enthalten. In diesem Fall bestimmt der Wert von $t1$ die Nummer des symbolischen Registers. Dessen Inhalt wird durch eine neue SMT-Variable ersetzt (Codezeile 13). Die SMT-Variablen sind so benannt, dass sie die Nummer des Registers sowie einen fortlaufenden Index enthalten (Codezeile 10).

Listing 2: symbolisches Register

```
1  case Opcode::ECALL: {
2    int expr1 = regs[RegFile::to].simplify().
       get_numeral_int();
3    if (expr1 == 3) {
4      std::string symbolicreg;
5      int regnumber = regs[RegFile::t1].simplify().
       get_numeral_int();
6      RegFile regfile;
7      //gloabler Index fuer sym reg
8      countsymRegs[regnumber]++;
9      //symbolisches Register mit Registernummer und
       globalem Index fuer das Auseinanderhalten bei
       kopierten Zustaenden
10     symbolicreg +=  "SymReg_x" + std::to_string(
       regnumber) + "_" + std::to_string(countsymRegs[
       regnumber]);
11     result = strdup(symbolicreg.c_str());
```

```
12      //erstellt z3 expressions
13      regs[regnumber] = contextZ3.bv_const(result, 32);
14    }
15  }
```

In Listing 3 wird die definierte Assumption verarbeitet. Bei einer Assumption würde *t*0 den Wert 2 enthalten. Zuerst wird der Solver zurückgesetzt, damit keine Bedingungen von vorherigen Anfragen das Ergebnis beeinflussen können. In Codezeile 8 wird die Pfadbedingung für die entsprechende Assumption aktualisiert. Hierbei wird an die bisherige Pfadbedingung der vereinfachte SMT-Ausdruck des Registers mit der Regnumber angehängt. Der Solver evaluiert, ob die Pfadbedingung erfüllt werden kann (Codezeile 10). Wenn die Assumption mit keiner Belegung erfüllbar ist, wird es an den Scheduler gemeldet, der die einzelnen Zustände verwaltet. Als Ausgabe gibt es eine Exception, die dem Scheduler mitteilt, dass die Assumption nicht gültig ist.

Listing 3: Assumption

```
1  case Opcode::ECALL: {
2    //Ist to := 2?
3    int expr1 = regs[RegFile::to].simplify().
       get_numeral_int();
4    if (expr1 == 2) {
5      //entfernt alles, was bisher in den Solver
       geschrieben wurde
6      sol.reset();
7      int regnumber = regs[RegFile::t1].simplify().
       get_numeral_int()];
8      pathCond = pathCond && regs[regnumber];
9      //Pfadbedingung an den Solver
10     sol.add(pathCond);
11     //Stelle ist mit keinen Input erreichbar, melde es
       dem Scheduler
12     if (solver_check(sol) == unsat) {
13       throw assumeexception;
14     }
15   }
16 }
```

Zum Schluss gibt es noch die Assertion in Listing 4. *t*0 würde hier den Wert 1 enthalten, wenn es sich um eine Assertion handelt. Bei einer Assertion wird immer *assert(Bedingung)* an den Sym-ISS gegeben. Bei der Pfadbedingung wird die negierte Assertion an die Pfadbedingung angehängt ($\pi \wedge \neg Bedingung$). Um zu testen, ob die Assertion verletzt wurde, wird erneut der SMT-Solver eingesetzt (Codezeile 9).

Wird die Assertion verletzt, gibt der SMT-Solver SAT zurück und es kann ein Gegenbeispiel aufgezeigt werden (Codezeile 11-15). Das bedeutet, dass es eine konkrete Belegung gibt, die die Assertion zu false auswerten lässt. Für den Benutzer wird das Model ausgegeben

(Codezeile 17). Das Model ist die Belegung aller Variablen, so dass die Assertion verletzt ist. In der Standardausgabe des Z3 werden Konstanten in Hexadezimal dargestellt. Für den User soll das Gegenbeispiel im Dezimalsystem angezeigt werden (Codezeile 14). Zusätzlich wird eine Exception geworfen, die dem Scheduler mitteilt, dass die Assertion nicht gültig ist.

Bei UNSAT wurde kein Fehler gefunden und die Assertion ist nicht verletzt. Durch die Negation wird der Teilausdruck dann wieder zu true. Der Teilausdruck für die Assertion wird außerdem nicht dauerhaft an die Pfadbedingung angehängt, wie es bei der Assumption der Fall ist (siehe Listing 4). Dies wäre zwar grundsätzlich möglich, würde allerdings die Pfadbedingung unnötig vergrößern und im weiteren Verlauf des Programms vermutlich wenig Mehrwert bringen.

Listing 4: Assertion

```
1  case Opcode::ECALL: {
2    //Ist to := 1?
3    int expr1 = regs[RegFile::to].simplify().
       get_numeral_int();
4    if (expr1 == 1) {
5      //entfernt alles, was bisher in den Solver
       geschrieben wurde
6      sol.reset();
7      int regnumber = regs[RegFile::t1].simplify().
       get_numeral_int()];
8      //Vereinfachter negierter SMT-Ausdruck der
       Registernummer von t1 verundet mit bisheriger
       Pfadbedingung
9      sol.add(pathCond && !(regs[regnumber]));
10     //assertion kann verletzt werden
11     if (sol.check() == sat) {
12       model m = sol.get_model();
13       for (int i = 0; i < symregs.size(); i++) {
14         long unsigned int result = hex2dec(m.eval(
       symregs[i]).to_string());
15         std::cout << "model eval " << symregs[i] << " "
       << result << std::endl;
16         //Model vom Gegenbeispiel ausgeben
17         showresult(m);
18       }
19       throw assertionexception;
20     }
21   }
22 }
```

Im Folgenden wird das Vorgehen anhand des Beispiels aus dem vorherigen Abschnitts erklärt.

```
1 #symbolisches Register = a0
2 li t1, 10 #RegNumber 10 = a0
3 li to, 3
```

```
4  ecall
5
6  #Assumption Bedingung, gespeichert in t2 (Zeile 12)
7  #t2:=a0 < 13
8  slti t2,a0,13
9  #a4:=0 < a0
10 slt a4,zero,a0
11 #t2:=t2 ∧ a4 = 0 < a0 < 13
12 and t2,t2,a4
13
14 #Assumptionaufruf
15 #RegNumber 7 = t2
16 li t1, 7
17 li t0,2
18 ecall
19
20
21 #a1:=a0+12
22 addi a1,a0,12
23
24 #Assertion Bedingung
25 #t2:=a1<24
26 slti t2,a1,24
27
28 #Assertionaufruf
29 #RegNumber 7 = t2
30 li t1, 7
31 li t0, 1
32 ecall
```

Die Pfadbedingung ist zu Beginn auf true gesetzt und ändert sich durch die Assumption in Codezeile 18 zu $\pi = (a0 < 13) \wedge (a0 > 0)$. Im nächsten Zustand findet die Addition mit 12 statt. Der SMT-Ausdruck für die Assertion lautet $\pi \wedge \neg(a1 < 24)$. Dieser Ausdruck wird in den Solver gegeben, sodass am Ende SAT oder UNSAT ausgegeben wird. Bei diesem Programm schlägt die Assertion fehl und es wird SAT mit dem Gegenbeispiel $a0 = 12 \wedge a1 = 24$ zurückgegeben.

3.4 *Anpassung der Instruktionen*

In dieser Arbeit wird die ISS aus der AG Rechnerarchitektur der Universität Bremen verwendet. Es gibt zum Beispiel Instruktionen für arithmetische Operationen, aber auch für logische Operatoren oder für die Speicherverwaltung. RV32I ist gekennzeichnet durch eine Registerbreite von 32 Bit und durch die entsprechende Größe des Benutzeradressraums. Beim Sym-ISS werden 32 Bit für alle Operationen verwendet. Alle Instruktionen müssen auf einer Vier-Byte-Grenze im Speicher ausgerichtet sein (siehe Waterman u. Asanovic̀ (2017)). Um

die Dekodierung zu beschleunigen, sind die wichtigsten Felder in jeder Anweisung im RISC-V an die gleiche Stelle gesetzt.

Da die gespeicherten Inhalte einiger Register symbolisch sein können, müssen die Instruktionen so angepasst werden, dass sowohl mit konstanten als auch mit symbolischen SMT-Ausdrücken weiter gerechnet werden kann. Dafür müssen alle Instruktionen, die vorher einen 32-Bit Integer zurückgegeben haben so angepasst werden, dass stattdessen SMT-Ausdrücke zurück gegeben werden. Zu Beginn müssen alle Register als SMT Bitvektoren initialisiert werden. Werden arithmetische Operationen auf den Registern ausgeführt, können diese ohne Schwierigkeiten vorgenommen werden.

```
1  #symbolisches Register = a0
2  li t1, 10 #RegNumber 10 = a0
3  li t0, 3
4  ecall
5
6  #Assumption Bedingung, gespeichert in t2 (Zeile 12)
7  #t2:=a0 < 13
8  slti t2,a0,13
9  #a4:=0 < a0
10 slt a4,zero,a0
11 #t2:=t2 ∧ a4 = 0 < a0 < 13
12 and t2,t2,a4
13
14 #Assumptionaufruf
15 #RegNumber 7 = t2
16 li t1, 7
17 li t0,2
18 ecall
19
20
21 #a1:=a0+12
22 addi a1,a0,12
23
24 #Assertion Bedingung
25 #t2:=a1<24
26 slti t2,a1,24
27
28 #Assertionaufruf
29 #RegNumber 7 = t2
30 li t1, 7
31 li t0, 1
32 ecall
```

In diesem Beispiel (wie aus dem vorherigen Abschnitt) wird bei der Addition ein SMT-Ausdruck des symbolischen Registers $a0$ mit dem konkreten Wert 12 addiert. Das heißt, aus dem SMT-Ausdruck und aus der konkreten Zahl werden Bitvektoren generiert, so dass sie miteinander addiert werden können. Der neue Wert des Registers $a1$ ist damit $a0 + 12$.

Es werden am Ende alle Instruktionen durch SMT-Ausdrücke umgesetzt (siehe Tabelle 14 in Anhang D). Da bei der Ausführung mehrerer Operationen die SMT-Ausdrücke sehr groß werden können, werden sie regelmäßig vereinfacht. Die Vereinfachung geschieht vor jedem Branch- und Set-Befehl und wird durch eine eingebaute Funktion des Z3-Solvers umgesetzt.

Im Folgenden werden einige Instruktionen exemplarisch genauer betrachtet. In den Tabellen 3 bis 8 befindet sich jeweils eine Abbildung ϕ von Register (Reg) auf SMT-Ausdrücke. Am Anfang steht der Name der jeweiligen Instruktion, gefolgt von der mathematischen Operation (Semantik) und des Befehls in Assembler (Syntax). In der letzten Spalte wird der entsprechende SMT-Ausdruck (siehe The SMT-LIB Initiative (2003)) angegeben. Das Zielregister ist definiert als r_d, *imm* steht für Immediate und *pcnt* ist der program counter. In allen SMT-Ausdrücken werden Bitvektoren der Länge 32 verwendet.

Tabelle 3: Abbildung für die Instruktion LUI

Name	Operation	Syntax	SMT-Ausdruck
LUI	$\$r_d$ = (imm $\ll$ 12)	lui $\$r_d$, imm	$\phi(\$r_d) = imm \ll 12$

LUI (Load Upper Immediate) (siehe Tabelle 3) platziert das Immediate an die Bitstellen $31 - 12$ und füllt die letzten 12 Bits mit Nullen auf. Bei dieser Instruktion werden die 12 Bits des Immediate nach links in das Zielregister geshiftet, indem 12 Nullen eingefügt werden.

Tabelle 4: Abbildung für die Instruktion JAL

Name	Operation	Syntax	SMT-Ausdruck
JAL	$\$r_d$ = pcnt + 4; pcnt = pcnt + imm	jal label	$\phi(\$r_d)$ = pcnt + 4

JAL (siehe Tabelle 4) ist ein Sprungbefehl und kann zu einer bestimmten Adresse im Speicher springen. Die Rücksprungadresse, also die Instruktion nach dem aktuellen program counter, wird im Zielregister gespeichert. Dabei wird der program counter erhöht, indem er mit dem Offset addiert wird.

Bei JALR (siehe Tabelle 5) springt man im Gegensatz zu JAL zu der Adresse eines Register. Aus diesem Grund berechnet sich der program counter durch die Summe aus dem Registerinhalt und dem Offset. Im Zielregister wird dann die entsprechende Adresse gespeichert.

SRAI (Shift Right Arithmetic Immediate) (siehe Tabelle 6) verschiebt die Bits um shamt nach rechts und speichert den Wert im Zielregister ab, wobei shamt Teil der Instruktion ist. In SMT ist $\gg$ wie die logische

Tabelle 5: Abbildung für die Instruktion JALR

Name	Operation	Syntax	SMT-Ausdruck
JALR	$\$r_d$ = pcnt + 4; pcnt = $\$r_{s1}$ + imm	jalr $\$r_d$, $\$r_{s1}$,label	$\phi(\$r_d)$ = pcnt + 4;

Tabelle 6: Abbildung für die Instruktion SRAI

Name	Operation	Syntax	SMT-Ausdruck
SRAI	$\$r_d = \$r_{s1} \gg$ shamt	srai $\$r_d$, $\$r_{s1}$, shamt	$\phi(\$r_d) = \phi(\$r_{s1}) \gg$ shamt

Verschiebung nach rechts, bei der die höchstwertigen Bits immer das höchstwertige Bit des ersten Arguments kopieren.

Tabelle 7: Abbildung für die Instruktion ADD

Name	Operation	Syntax	SMT-Ausdruck
ADD	$\$r_d = \$r_{s1} + \$r_{s2}$	add $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} + \$r_{s2})$

Bei ADD (siehe Tabelle 7) werden die Inhalte von zwei Registern addiert. In SMT werden dafür Bitvektoren benutzt, wodurch automatisch ein Overflow modelliert wird. Das Ergebnis wird im Zielregister gespeichert.

Tabelle 8: Abbildung für die Instruktion XOR

Name	Operation	Syntax	SMT-Ausdruck
XOR	$\$r_d = \$r_{s1} \wedge \$r_{s2}$	xor $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} \wedge \$r_{s2})$

XOR (siehe Tabelle 8) ist ähnlich zur Addition. Hier werden die einzelnen Bits nur nicht miteinander addiert, sondern es wird ein bitweises Exklusiv-Oder verwendet. Das Ergebnis wird wieder im Zielregister gespeichert.

3.5 *Modellierung des Speichers*

Der Sym-ISS greift nicht nur auf Register zu, sondern nimmt bei *load*- und *store*-Befehlen ebenfalls Zugriffe auf den Hauptspeicher vor. In

Tabelle 9 werden die beiden Instruktionen mit ihren entsprechenden SMT-Ausdrücken einmal aufgelistet.

Tabelle 9: Abbildung für die Instruktion LB und LW

Name	Operation	Syntax	SMT-Ausdruck
LB	$\$r_d$ = mem[$\$r_{s1}$ + imm]	lb $\$r_d$, imm($\$r_{s1}$)	$\phi(\$r_d)$=$\phi$(mem[$\$r_{s1}$ + imm])
LW	$\$r_d$ = mem[$\$r_{s1}$ + imm … $\$r_{s1} + imm + 3$]	lw $\$r_d$, imm($\$r_{s1}$)	$\phi(\$r_d)$=$\phi$(mem[$\$r_{s1}$ + imm … $\$r_{s1} + imm + 3$])

LB (Load Byte) lädt ein Byte aus dem Hauptspeicher. Das Byte aus dem Speicher wird in die acht niederwertigen Bits des Registers gespeichert. LW (Load Word) lädt ein Wort aus dem Hauptspeicher. Hierfür wird das gleiche Vorgehen wie bei LB verwendet, nur dass insgesamt 4 Byte geladen werden. Dies sind die Speicheradressen von $\$r_{s1} + imm$ bis $\$r_{s1} + imm + 3$.

Eine naive Möglichkeit den Speicher symbolisch umzusetzen, wäre es, den Hauptspeicher genau wie die Register vollständig symbolisch zu behandeln. Dies ist jedoch aufgrund der Größe des Hauptspeichers nicht praktikabel. Daher enthalten die Speicherzellen vorerst konkrete Werte und werden nur bei Bedarf durch SMT-Ausdrücke ersetzt.

Im Sym-ISS wird zu den bisherigen *ECALL*s der *ECALL* für den symbolischen Hauptspeicher hinzugefügt (siehe Listing 5). Es werden an der entsprechenden Adresse die vorgegebene Anzahl an Bytes symbolisch definiert. An der entsprechenden symbolischen Bytestelle wird der jeweilige SMT-Ausdruck gespeichert.

Listing 5: symbolischer Hauptspeicher

```
1  case Opcode::ECALL: {
2    //Ist to := 4?
3    int expr1 = regs[RegFile::to].simplify().
       get_numeral_int();
4    if (expr1 == 4) {
5      uint32_t addr = regs[RegFile::t1].simplify().
       get_numeral_uint();
6      uint32_t num_bytes = regs[RegFile::t2].simplify().
       get_numeral_uint();
7      //Markiert entsprechende Stellen der Adresse
       symbolisch und bildet sie auf die Stellen im
       Hauptspeicher ab
8      for(uint32_t i = addr; i < addr + num_bytes; i++){
9        int counter;
10       auto it = countsymBytes.find(i);
11       if (it == countsymBytes.end()){
12         counter = 0;
13         countsymBytes.insert(std::pair<uint32_t,
       uint32_t>(i,counter));
14       }else{
```

```
15          counter = ++it->second;
16        }
17        std::string symbolicbyte = "Sym_Byte" + std::
      to_string(i) + "_" + std::to_string(counter);
18        z3::expr e = contextZ3.bv_const(symbolicbyte.
      c_str(), 32);
19        //SMT-Ausdruck wird im Hauptspeicher gespeichert
20        mem->_store_single_byte(i, e);
21      }
22    }
23 }
```

Bei einem symbolischen Hauptspeicher würde $t0$ den Wert 4 enthalten (siehe Listing 5). Zu Beginn werden die Werte der Register $t0$, $t1$ und $t2$ abgefragt. In Register $t1$ ist die Adresse der Zelle im Hauptspeicher hinterlegt, die symbolisch werden soll (Codezeile 5). Der Wert von $t2$ bestimmt die Anzahl der Bytes, die von der Zelle symbolisch werden sollen (Codezeile 6). Im Hauptspeicher werden die entsprechenden Stellen der Adresse symbolisch markiert und auf die entsprechende Stelle im konkreten Hauptspeicher abgebildet (Codezeile 18). Dessen Inhalt wird durch eine neue SMT-Variable ersetzt und im Hauptspeicher gespeichert (Codezeile 20). Die SMT-Variablen sind so benannt, dass sie die Adresse des Hauptspeichers sowie einen fortlaufenden Index enthalten (Codezeile 17).

3.6 *Integration in C-Programmen*

Durch die Veränderungen am Speicher ist es nur ein kleiner weiterer Schritt, um C-Programme zu unterstützen. Dafür müssen die Befehle für Assertions und Assumptions auf C-Ebene gebracht werden. Daher wird ein Befehl benötigt, der Assembleranweisungen in C/C++-Quellcode wrappt. Im Folgenden wird hierfür der Befehl *asm("Assembler")* verwendet (siehe Listing 6). Auf diese Weise entsteht die Möglichkeit, hardwarenahe Aufgaben wie den Umgang mit Hardwareregistern in einem C-Programm zu erledigen. Ein weiterer Hinweis für den Compiler ist das Schlüsselwort *register* (zum Beispiel in Codezeile 2). Damit wird ausgedrückt, dass eine Variable in einem Register gespeichert werden soll. Somit ist es möglich, die gleiche Struktur wie bei dem vorherigen *ECALL* in Assembler beizubehalten.

Ein Unterschied zu Assembler ist der Aufruf der Methode in C. Damit die Assumptions und Assertions sinnvoll übergeben werden können, enthalten die Methoden diese als Parameter (Codezeile 1 und 8). Die Assumptions und Assertions werden nach dem gleichen Vorgehen wie in Assembler in Register $t2$ gespeichert (Codezeile 4 und 11). Das entsprechende Register ist intern Registernummer 7. Deshalb wird in Register $t1$ die entsprechende Registernummer angegeben.

Listing 6: Ecalls

```
1  void riscv_assume(int cond) {
2    register long to asm("to") = 2;
3    register long t1 asm("t1") = 7;
4    register long t2 asm("t2") = cond;
5    asm volatile ("ecall");
6  }
7
8  void riscv_assert(int cond) {
9    register long to asm("to") = 1;
10   register long t1 asm("t1") = 7;
11   register long t2 asm("t2") = cond;
12   asm volatile ("ecall");
13 }
```

Zusätzlich wurde die Instruktion *ECALL* mit einem Parameter erweitert, mit dem ein symbolischer Hauptspeicher generiert werden kann (siehe Listing 7). Hierfür muss der Wert in Register $t0$ 4 sein, damit ein symbolischer Hauptspeicher erstellt werden kann (Codezeile 2).

Listing 7: Ecall für symbolischen Hauptspeicher

```
1  void riscv_make_symbolic (unsigned int addr, unsigned
       int num_bytes){
2    register long to asm("to") = 4;
3    register long t1 asm("t1") = addr;
4    register long t2 asm("t2") = num_bytes;
5    asm volatile ("ecall");
6  }
```

Mit dem Aufruf dieser Methode im Eingabeprogramm kann bei einer bestimmten Adresse eine bestimmte Anzahl an Bytes symbolisch gemacht werden. In Register $t1$ werden die Adresse und in Register $t2$ die Anzahl der Bytes definiert, die symbolisch werden sollen.

Anhand eines Beispielprogramms (siehe Listing 8) wird verdeutlicht, wie ein Teil des Hauptspeichers symbolisch gemacht werden kann.

Listing 8: Beispiel eines Eingabeprogramms in C++

```
1  //Methoden fuer riscv_make_symbolic, riscv_assume und
       riscv_assert siehe Listing 6,7
2  typedef unsigned int uint;
3  int sum (int* field, uint size){
4    int sum = 0;
5    for(uint i = 0; i < size; i++){
6      sum += field[i];
7    }
8    return sum;
9  }
10
11 int main(){
12   uint max_size = 10;
```

```
13    int size;
14    riscv_make_symbolic(&size,sizeof(size));
15    riscv_assume(size <= max_size);
16    riscv_assume(size >= 1);
17    int field[max_size];
18    for(uint i = 0; i < max_size; i++){
19      riscv_make_symbolic(&field[i],sizeof(field[i]));
20      riscv_assume(field[i] >= 0);
21  }
22
23  riscv_assert(sum(field, size) >= 0);
24  return 0;
25  }
```

In dem Beispiel werden die Elemente eines Arrays miteinander addiert. Es gibt einmal die Methode *sum*, die die Elemente des Arrays entsprechend addiert (Codezeile 3-9). In der *main*-Methode werden unter anderem die Assumptions, Assertions und der symbolische Hauptspeicher definiert. Der Hauptspeicher wird an der Stelle 1 bis 10 des Arrays symbolisch gemacht (Codezeile 14-16). Das Array wird dann mit der entsprechenden Größe erstellt und die entsprechenden Felder werden symbolisch. Dabei soll jedes Feld im Array ≥ 0 sein. Die Assertion erwartet eine Summe ≥ 0 (Codezeile 23).

In diesem Beispiel schlägt die Assertion fehl, da die Summe überläuft.

4 BRANCHING

Bisher wurden in dieser Arbeit nur Eingabeprogramme vorgestellt, die kein Branching enthalten haben. In den folgenden Abschnitten wird beschrieben, wie das Branching in der symbolischen Ausführungs-Engine realisiert wird (siehe Abschnitt 4.1).

Sobald ein Branching zu einer Kopie eines Zustandes führt, spricht man vom Clonen (siehe Abschnitt 4.2). Für die Auswahl des zu bearbeitenden Zustandes wählt der Scheduler diesen anhand einer Explorationsstrategie. Diese Strategien werden in Abschnitt 4.3 beschrieben.

4.1 *Allgemeines Vorgehen beim Branchen*

If-Bedingungen führen zu einem Branching. Auf Assemblerebene wird dieses Konstrukt zum Beispiel durch die Instruktion *blt* umgesetzt. In Tabelle 10 wird die Instruktion abgebildet.

Tabelle 10: Abbildung für die Instruktion BLT

Name	Operation	Syntax	SMT-Ausdruck
BLT	if $\$r_{s1}$ < $\$r_{s2}$ then pcnt = pcnt + imm	blt $\$r_{s1}$, $\$r_{s2}$, label	-

BLT (Branch Less Than) überprüft, ob der Inhalt des einen Registers kleiner als der Inhalt des anderen Registers ist. In diesem Fall wird zur Adresse gesprungen. Da hier nur der program counter aktualisiert wird, werden keine neuen SMT-Ausdrücke benötigt.

Das Eingabeprogramm in C wird im Sym-ISS auf Assemblerebene bearbeitet. Das bedeutet, dass eine if-Bedingung als Instruktion

$$bxx\ \$r_d, \$r_{s1}, label$$

dargestellt werden kann. *bxx* steht dabei für einen beliebigen Branch-Befehl wie *blt* oder *beq*. Innerhalb dieser Instruktion wird dann der Zustand einmal geclont. Bei einem *bxx* gibt es daher eine Bedingung *cond*, die zu true oder false ausgewertet werden kann. Der Ausführungspfad S wird daher in zwei unäbhängige Pfade S_{true} und S_{false} aufgeteilt. Die Pfadbedingung wird für jeden Pfad entsprechend aktualisiert zu $\pi(S_{true}) = \pi(S) \wedge cond$ bzw. $\pi(S_{false}) = \pi(S) \wedge \neg cond$. Der SMT-Solver kann herausfinden, ob beide Pfade möglich wären. Es muss einen Zustand geben, der beim *label* weitermacht und einen Zustand der den bisherigen Pfad weiter abarbeitet.

Wenn die Bedingung des else-Teils SAT ist, dann wird der program counter auf den nächsten Befehl gesetzt, da das Eingabeprogramm

weiter abgearbeitet werden kann. Ist die Bedingung des then-Teils SAT, dann wird gesprungen und der program counter wird angepasst.

Angenommen, es gibt einen Zustand mit einem program counter von 10088 und eine Pfadbedingung mit $\pi = (a0 < 12) \wedge (a0 > 5)$. Es gibt das symbolische Register $a0$ und ein weiteres Register $a1 := 9$. Der abzuarbeitende Befehl lautet $blt\ a0, a1, 10080$. Im Folgenden wird der Zustand geclont. Das heißt, die Pfadbedingung, der program counter, die Register und der Hauptspeicher werden wie oben beschrieben kopiert. Die eine Pfadbedingung wird zu $\pi = (a0 < 12) \wedge (a0 > 5) \wedge (a0 < 9)$ und die andere zu $\pi = (a0 < 12) \wedge (a0 < 5) \wedge (a0 \geq 9)$ geändert. Der program counter ändert sich von dem ersten Zustand zu 100080 und bei zweiten Zustand bleibt der program counter 10088. Die Register und der Hauptspeicher ändern sich nicht. Der SMT-Solver prüft beide Pfadbedingungen und gibt bei beiden Zuständen SAT zurück.

4.2 *Clonen der Zustände*

Beim Clonen eines Zustandes wird ein neuer Zustand erstellt. Dieser soll die gleichen Eigenschaften haben wie der ursprüngliche Zustand. Das bedeutet, dass sowohl die Pfadbedingung, der program counter, die Register und der Hautpspeicher kopiert werden müssen. Aus Performanzgründen werden nicht alle Bestandteile vollständig kopiert. Teilweise genügt es auch, dass der neue Zustand einen Pointer auf den entsprechenden Bereich des aktuellen Zustands hat.

Die Register werden vollständig kopiert. Das bedeutet, dass die Registerbelegung des aktuellen Zustands komplett kopiert wird und der neue Zustand dann die entsprechenden Register mit neuen Werten belegen kann, ohne den vorherigen Zustand zu beeinflussen. Die Pfadbedingung und der program counter werden ebenfalls vollständig kopiert. Da es sich hier um einen Integerwert und einen SMT-Ausdruck handelt, beansprucht dieser Kopiervorgang nicht viel Speicherplatz. Beim Hauptspeicher ist eine vollständige Kopie aufgrund der Größe nicht sinnvoll. Daher wird für den Hauptspeicher eine persistente Datenstruktur verwendet. Das bedeutet, dass alte Zustände des Speichers erhalten bleiben und nur Bereiche, in denen Änderungen vorkommen, explizit vermerkt werden. Hierdurch ist eine schnelle Kopie des Speichers möglich, indem nur ein Pointer kopiert wird.

Sobald ein Branching zum Clonen eines Zustandes führt, wird die Pfadbedingung des Zustandes durch den SMT-Solver evaluiert. Alle kopierten Zustände, bei denen die Pfadbedingung SAT ist, werden durch einen Scheduler verwaltet. Zustände mit einer unerfüllbaren Pfadbedingung werden verworfen.

Der Scheduler wählt anhand von Explorationsstrategien den nächsten abzuarbeitenden Zustand aus.

4.3 *Explorationsstrategien*

In dieser Arbeit werden zwei Explorationsstrategien implementiert. Die erste Strategie verfolgt den naiven Ansatz, die Zustände nacheinander von einem Stack abzuarbeiten. Der erste Zustand aus dem Stack wird durch den Sym-ISS bearbeitet. Ist die Assumption mit keinem Input erreichbar, wird der nächste Zustand vom Stack geholt. Dieser Ansatz entspricht einer Tiefensuche auf dem Execution Tree und ist die gängige Strategie in vielen symbolischen Ausführungs-Engines wie zum Beispiel bei DART (siehe Baldoni u. a. (2018)).

Wenn das Eingabeprogramm eine Schleife enthält (siehe Listing 9), dann kann in folgender Situation ein Problem auftreten.

Listing 9: Eingabeprogramm

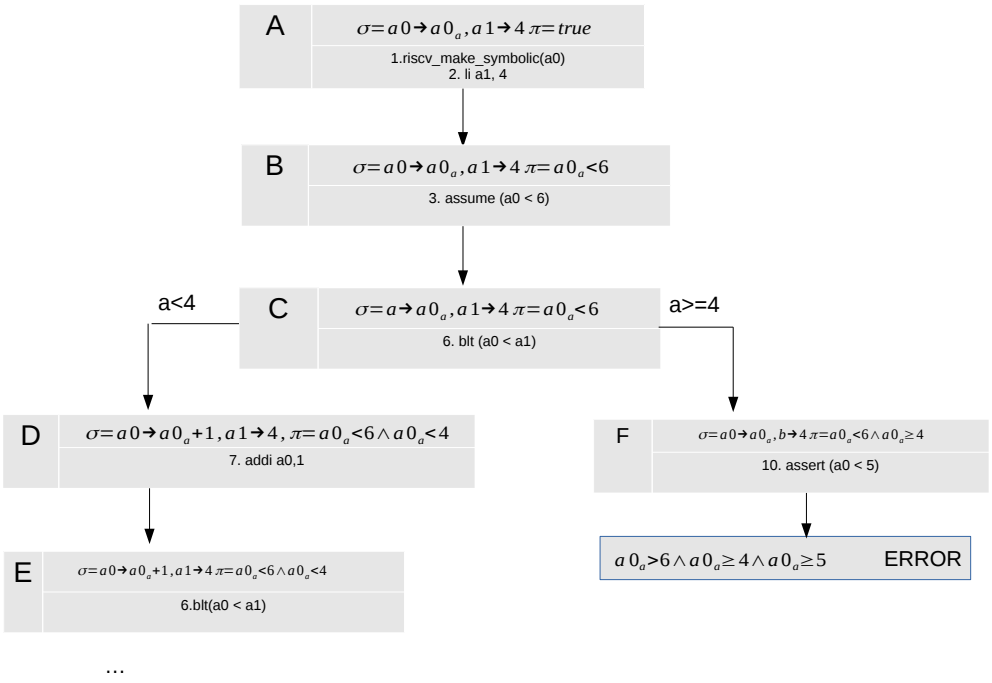
```
1  //Methoden fuer riscv_make_symbolic, riscv_assume und
       riscv_assert siehe Listing 6,7
2  int main(){
3    int symb;
4    riscv_make_symbolic(&symb,sizeof(symb));
5    riscv_assume(symb < 7);
6    while(4 < symb){
7      symb += 1;
8    }
9    riscv_assert(symb < 6);
10   return 0;
11 }
```

Es gibt eine symbolische Variable *symb*. Die Assumption für das Programm ist definiert als $(symb < 7)$. Die Assertion definiert, dass nach Ausführung der Schleife $symb < 6$ sein soll. Bei der while-Schleife wird es einen Zustand geben für $1)(4 < symb)$ und einen für $2)(4 \geq symb)$. Beide Zustände werden auf den Stack gelegt. Ein Problem bei dem Abarbeiten des Stacks ist, dass 2) nicht ausgeführt wird, solange 1) erreichbar ist. Das liegt daran, dass jedes Mal vorher in 1) gegangen wurde und wieder zwei neue Zustände auf den Stack gelegt werden.

Wenn zuerst der Zustand von 1) bearbeitet wird, dann wird $a0$ um 1 erhöht, wenn $a0 < 4$. Im Anschluss daran wird wieder zum *blt* gesprungen und es werden wieder zwei neue Zustände auf den Stack gelegt. Zuerst wird der Zustand von 1) und dann der Zustand von 2) auf den Stack gelegt. Somit wird $a0$ wieder um eins erhöht, wenn $a0 \leq 4$. Es kommen immer zwei neue Zustände dazu, so lange bis in 2) gegangen wird.

Bei dem Aufbau von Programmen wie bei diesem Beispiel, würden bei der Tiefensuche immer nur die Zustände aus dem linken Pfad

Abbildung 5: Execution Tree für das Eingabeprogramm



des Execution Trees ausgewählt werden und der Sym-ISS würde nie den Zustand *F* wählen (siehe Abbildung 5). Somit führt diese Explorationsstrategie zu sehr langen Laufzeiten.

Eine andere mögliche Strategie wäre es, einen Zustand zufällig zu wählen. Zum Beispiel könnte zufällig der Zustand von 2) nach dem dritten Schleifendurchlauf gewählt werden. Das führt dazu, dass aus der Schleife herausgesprungen wird und das Programm beendet ist. Eine zufällige Auswahl des Zustandes führt dazu, dass der Execution Tree gleichmäßig aufgebaut ist und der Sym-ISS sich nicht nur in einem Pfad aufhält, sondern auch einen Zustand von einem ganz anderen Pfad auswählt.

Bei der Auswahl eines zufälligen Zustandes wird früher oder später auf jeden Fall auch einmal der Zustand auf dem rechten Pfad (*F*) gewählt und das Programm endet somit in einer adäquaten Laufzeit.

5 EVALUATION

In diesem Abschnitt wird die entwickelte symbolische Ausführung der RISC-V-ISA evaluiert. Zunächst werden notwendige Eingabeprogramme vorgestellt, die für die Evaluation in C geschrieben worden sind. Es handelt sich dabei um Programme, die leicht skaliert werden können und unterschiedliche Konstrukte enthalten, wie zum Beispiel Arrays, Schleifen, if-Bedingungen oder Strings.

Die in dieser Arbeit entwickelte symbolische Ausführung wurde in C++ geschrieben und es wird in der Evaluation untersucht, ob die Eingabeprogramme korrekt verifiziert werden und der SMT-Solver das richtige Ergebnis zurück gibt. Zusätzlich werden die in der symbolischen Ausführungs-Engine realisierten Explorationsstrategien gegenüber gestellt und es wird untersucht, wie viele Pfade jeweils generiert werden, um die eingebauten Fehler im Eingabeprogramm zu finden.

Die untersuchten Explorationsstrategien sind einmal eine Tiefensuche und die zufällige Auswahl des nächsten Zustandes. Die Tiefensuche ist eine weit verbreitete mögliche Strategie, um den nächsten Zustand auszuwählen. Sie wird zum Beispiel bei der symbolischen Ausführungs-Engine *DART* verwendet (siehe Baldoni u. a. (2018)).

Bei der zufälligen Auswahl eines neuen Zustandes wird sicher gestellt, dass nicht derselbe Pfad verfolgt wird, sondern auch andere Pfade ausgewählt werden können.

5.1 *Eingabeprogramme*

Es wurden insgesamt 11 unterschiedliche Eingabeprogramme untersucht, bei denen eine Assertion fehlschlägt und 10 Eingabeprogramme, bei denen Assertions halten. Es gibt ein zusätzliches Eingabeprogramm, bei der eine Assertion fehlschlägt, da hiermit besonders demonstriert werden soll, dass die Explorationsstrategien ein gewisses Verhalten bewirken.

Die Eingabeprogramme wurden selbst entwickelt, damit sie jederzeit verändert werden können. Es sind bewusst nicht die Beispielprogramme vom Test Repository verwendet wurden, da diese sehr viele externe Abhängigkeiten, insbesondere Bibliotheken enthalten.[1]

Die folgenden Funktionen werden durch die Eingabeprogramme evaluiert:

- Alle Felder innerhalb eines Arrays miteinander addieren
- If-Bedingungen, die einen Branch herbei führen
- Die Sortieralgorithmen Bubblesort, Quicksort und Mergesort auf Arrays

1 Dies hätte den Rahmen der Masterarbeit gesprengt.

- Programme, um Unterschiede zwischen den Explorationsstrategien herauszustellen
- Fibonacci (rekursiv und iterativ)
- Maximum in einem Array finden
- Schnitt von zwei Mengen (repräsentiert durch Arrays)
- String nach Int umwandeln

Im Folgenden wird auf einige spezielle Eingabeprogramme näher eingegangen.

Listing 10: Auszug Bubblesort

```
1  //Zu den Methoden riscv_make_symbolic, riscv_assume und
       riscv_assert siehe Listing 6,7
2
3  void bubblesort(int *array, int length){
4    int i, j,tmp;
5    //Korrekte Zeile: for (i = 1; i < length; ++i)
6    for (i = 1; i < length - 1 ; ++i){
7      for (j = 0; j < length - i ; ++j){
8        if (array[j] > array[j+1]){
9          tmp = array[j];
10         array[j] = array[j+1];
11         array[j+1] = tmp;
12       }
13     }
14   }
15 }
16
17 int main(){
18   unsigned int max_size = 7;
19   int size;
20   riscv_make_symbolic(&size,sizeof(size));
21   riscv_assume(size <= max_size);
22   riscv_assume(size >= 1);
23   int array[max_size];
24   for(int i = 0; i < max_size; i++){
25     riscv_make_symbolic(&array[i],sizeof(array[i]));
26   }
27   bubblesort(array,size);
28   for(int i = 0; i < size-1; i++){
29     riscv_assert(array[i] <= array[i+1]);
30   }
31 }
```

Der Bubblesort-Algorithmus ist in der Methode *bubblesort* implementiert (siehe Listing 10). Elemente im Array werden getauscht, wenn das nächste Element im Array kleiner ist als das aktuelle Element (Codezeile 8-11). In der Methode *main* sind sowohl die Größe des

Arrays als auch die Einträge des Arrays symbolisch. Dabei ist die Größe durch Assumptions so beschränkt, dass sie zwischen 1 und max_size liegt (Codezeile 21 und 22). Die Assertion überprüft, ob jedes Element im Array kleiner gleich dem nächsten Element des Arrays ist. Das heißt, ob das Array korrekt sortiert ist (Codezeile 29). Durch einen Kommentar ist die Stelle markiert, an der ein Fehler im Programm eingebaut worden ist. Dieser Fehler führt dazu, dass die Assertion fehlschlägt (Codezeile 5). Angenommen, es gibt das Array (3,2,1), welches durch den Bubblesort-Algorithmus sortiert werden soll, dann bricht der Algorithmus nach der ersten Iteration ab. Das Array ist aber erst nach der zweiten Iteration vollständig sortiert. Die Ausgabe ist nach der ersten Iteration (2,1,3) und somit schlägt die Assertion fehl.

Als Nächstes wird ein konstruiertes Programm vorgestellt, bei dem der Execution Tree sehr unausgeglichen ist und somit die Explorationsstrategie einen starken Einfluss auf die Ausführungszeit nehmen kann (siehe Listing 11).

Listing 11: Auszug konstruiertes Programm

```
1 //Zu den Methoden riscv_make_symbolic, riscv_assume und
      riscv_assert siehe Listing 6,7
2
3 int main(){
4 int a;
5 riscv_make_symbolic(&a,sizeof(a));
6 for(;a !=0;a++);
7 riscv_assert(0);
8 }
```

In diesem Eingabeprogramm ist der Inputinteger a symbolisch (Codezeile 5) und er wird so lange erhöht, wie $a \neq 0$ ist. Das greift das in Abschnitt 4.3 geschilderte Problem auf. Die symbolische Variable a kann auch negative Werte annehmen, so dass es bei der Schleife zwei Fälle gibt. Entweder wird die Schleife verlassen oder sie wird erneut durchlaufen. Bei der Tiefensuche wird bei diesem Beispiel sehr lange in der Schleife geblieben und es werden immer wieder neue Zustände gebildet.

Wenn allerdings der Zustand zufällig ausgewählt wird, wird die Wahrscheinlichkeit erhöht, dass aus der Schleife herausgesprungen wird. Hierdurch wird die Assertion erreicht und das Programm beendet.

Das letzte Eingabeprogramm, dass vorgestellt wird, ist die Berechnung des Schnitts von zwei Mengen (siehe Listing 12).

Listing 12: Auszug von dem Schnitt von zwei Mengen

```
1
2  int intersection(int array1[], int array2[], int size1,
       int size2, int result[]){
3    int l = 0;
4    for (int i = 0; i < size1; i++){
5      for (int j = 0; j < size2; j++){
6        if(array1[i] == array2[j]){
7          //++l statt l++
8          result[++l] = array1[i];
9        }
10     }
11   }
12 return l;
13 }
14
15 int contains(int ar[], int sz, int e){
16   for (int i = 0; i < sz; i++){
17     if (ar[i] == e){
18       return 1;
19     }
20   }
21   return 0;
22 }
23
24 int main(){
25   unsigned int max_size = 8;
26   int size1;
27   riscv_make_symbolic(&size1,sizeof(size1));
28   riscv_assume(size1 <= max_size);
29   riscv_assume(size1 >= 0);
30   int array1[max_size];
31   int size2;
32   riscv_make_symbolic(&size2,sizeof(size2));
33   riscv_assume(size2 <= max_size);
34   riscv_assume(size2 >= 0);
35   int array2[max_size];
36   for(int i = 0; i < max_size; i++){
37     riscv_make_symbolic(&array1[i],sizeof(array1[i]));
38     riscv_assume(!contains(array1, i, array1[i]));
39   }
40   for(int i = 0; i < max_size; i++){
41     riscv_make_symbolic(&array2[i],sizeof(array2[i]));
42     riscv_assume(!contains(array2, i, array2[i]));
43   }
44   int result[max_size];
45   int result_size = intersection(array1, array2, size1,
       size2, result);
46   for (int i = 0; i < size1; i++){
47     riscv_assert(contains(array2, size2, array1[i]) ==
      contains(result, result_size, array1[i]));
```

```
48    }
49    for (int i = 0; i < size2; i++){
50      riscv_assert(contains(array1, size1, array2[i]) ==
       contains(result, result_size, array2[i]));
51    }
52    for(int i = 0; i < result_size; i++){
53      riscv_assert(!contains(result, i, result[i]));
54    }
55  }
```

In diesem Eingabeprogramm gibt es zwei symbolische Arrays, wo wieder sowohl die Größe der Arrays als auch alle Einträge symbolisch sind. Die Größe ist dabei durch Assumptions beschränkt, so dass sie zwischen 1 und max_size liegt (Codezeile 28,29 und 33,34). Die Methode *intersection* bildet den Schnitt von beiden Arrays. Die Assumptions stellen sicher, dass die beiden Arrays keine Duplikate enthalten, das heißt, dass sie sich wirklich wie Mengen verhalten. Die Assertions stellen sicher, dass ein Element genau dann im Schnitt enthalten ist, wenn es auch in beiden Arrays ist. Außerdem wird verlangt, dass das Ergebnis wiederum keine Duplikate enthält.

Das Vorgehen wiederholt sich in allen Eingabeprogrammen. Es ist immer sowohl die Größe des Arrays als auch die Einträge im Array symbolisch. Bei einem Integer ist immer der Integer symbolisch und ein char* bei *StringToNumber* wird wie ein Array behandelt.

5.2 *Ergebnisse und Auswertung*

Die Evaluation wurde auf einem Intel Xeon CPU E3 – 1240 V2 mit 3,4~GHz, 8 Kernen und einem 32 GB Hauptspeicher ausgeführt. Als SMT-Solver wurde der Z3-Solver (Version 4.8.0) eingesetzt (siehe Bjørner u. de Moura (2008)). In den Tabellen 11 und 12 werden die Ergebnisse der Auswertung präsentiert. Da jedes Eingabeprogramm einmal fehlerfrei und mit einem Fehler versehen getestet wird, gibt es einmal eine Tabelle der Ergebnisse für fehlerbehaftete Programme (Tabelle 11) und eine für fehlerfreie (Tabelle 12) Programme.

In Tabelle 11 wird zu jedem Programm das Ergebnis (Assertion failed), die Codezeilen des Eingabeprogrammes in Assembler dargestellt und angegeben, welches Element bis zu welcher Größe im Eingabeprogramm symbolisch ist. Wie schon im vorherigen Abschnitt erwähnt, wurden die beiden Algorithmen Tiefensuche und Random untersucht. Hierbei wird jeweils die Anzahl der Zustände und die benötigte Laufzeit in Sekunden angegeben.

Es ist zu erkennen, dass die Laufzeit mit der Tiefensuche oftmals höher ist als wenn ein Zustand zufällig ausgewählt wird. In 10 von 11 Fällen ist die Tiefensuche langsamer. Entstehen mit der Tiefensuche weniger als 10 Zustände, so ist der Unterschied in der Ausführungszeit nicht so signifikant. Besonders interessant sind die Ergebnisse bei dem

Eingabeprogramm *Constructed*. Hier ist der Fall aufgetreten, dass bei der Tiefensuche keine Lösung innerhalb des gesetzten Timeouts von 24 Stunden gefunden werden konnte. Das liegt daran, dass dieses Programm das Problem in Abschnitt 4.3 verdeutlicht. Entsprechend sind auch immer die Anzahl der Zustände bei der Tiefensuche erheblich größer als beim zufälligen Auswählen eines Zustandes.

Es lässt sich sagen, das beim Testen der Eingabeprogramme immer das erwartete Ergebnis heraus gekommen ist.

Da die Explorationsstrategien nur eine Auswirkung auf Programme haben, bei denen die Assertion fehl schlägt, werden die Ergebnisse der fehlerfreien Programme in Tabelle 12 dargestellt. Hierbei bedeutet safe, dass die Assertion nicht fehlgeschlagen ist.

Tabelle 11: Ergebnisse für Tiefensuche und Random (unsafe)

				Tiefensuche		Random	
Programm	E	#CZ	symZ	#Z	Zeit	#Z	Zeit
Addieren	U	223	Array($\leq$ 10)	11	6.71	8	6.57
Branch	U	105	Integer($\geq$ 8)	3	6.24	4	6.22
BubbleSort	U	275	Array($\leq$ 7)	22	10.98	7	6.56
Constructed	U	97	Integer	TO	TO	3	6.10
Fibonacci1	U	195	Integer($\leq$ 15)	16	86.51	3	6.12
Fibonacci2	U	194	Integer($\leq$ 15)	16	86.81	7	6.30
Loop	U	106	Loopcounter($\leq$ 450)	452	697.85	3	6.04
Max	U	254	Array($\leq$ 15)	46	12.00	22	10.81
MergeSort	U	461	Array($=$ 7)	40	15.69	99	26.91
QuickSort	U	312	Array($\leq$ 10)	4	8.77	5	8.78
Sets	U	537	Arrays($\leq$ 8)	147	51.94	153	17.15
StringToNum	U	426	char*($\leq$ 10)	589	39689.70	227	12.25

Legende:

E – Ergebnis

U – unsafe (Assertion failed)

#CZ – Anzahl Assemblercodezeilen des Eingabeprogrammes

symZ – Was ist symbolisch?

#Z – Anzahl Zustände

TO – Timeout bei $>$ 24 Stunden

Zeit – Laufzeit in s

Die Auflistung der Ergebnisse in Tabelle 12 ist ebenfalls sehr interessant, da überprüft werden kann, ob bei beiden Verfahren die gleiche Anzahl an Zuständen benötigt werden und die Laufzeiten sich nur gering unterscheiden. Die Anzahl der Zustände müssen gleich sein, da der gesamte Execution Tree durchsucht werden muss und dieser unabhängig von der Explorationsstrategie den gleichen Aufbau hat. Dies ist ebenfalls der Grund, warum die Laufzeiten sich bei beiden Strategien ähneln sollten.

Allgemein lässt sich sagen, dass ca 75% der Laufzeiten vom SMT-Solver beansprucht werden. Die restliche Zeit wird hauptsächlich für das Bilden der SMT-Ausdrücke benötigt. Bei größeren Benchmarks ist der Anteil der Solver-Zeit tendenziell sogar noch höher.

Anhand dieser Evalutation lässt sich feststellen, dass der Sym-ISS bezüglich der hier gewählen Vor- und Nachbedingungen korrekt ist, da die Vorbedingung für jede Eingabe erfüllt ist. Wenn das Programm terminiert, erfüllt das Ergebnis die Nachbedingung, bei einer Assertion, die nicht fehlschlägt. Soll die Assertion fehlschlagen, dann kann das Ergebnis auch nicht die Nachbedingung erfüllen.

Da bei safe der komplette Execution Tree durchsucht werden muss, sollten die Laufzeiten bei den Eingabeprogrammen mit dem Ergebnis *safe* langsamer sein als die Eingabeprogramme mit dem Ergebnis *unsafe*. Dies stimmt auch immer, außer bei dem Eingabeprogramm *StringToNum*. Hier wurde das Eingabeprogramm dahingehend verändert, dass es zwei Strings gibt und nicht nur einen String, wie bei dem Eingabeprogramm mit dem Ergebnis *unsafe*. Deshalb ist das Eingabeprogramm mit dem Ergebnis *safe* komplexer und weist eine höhere Laufzeit auf.

In der Tabelle 13 ist exemplarisch für die Sortieralgorithmen der Bubblesort skaliert worden. Ab einer Arraygröße von 10 kann innerhalb von 48 Stunden mit der Tiefensuche keine Lösung gefunden werden. Allgemein lässt sich sagen, dass die Anzahl der Zustände sich mit der Größe des Arrays startk erhöht.

Tabelle 12: Ergebnisse für Tiefensuche und Random (safe)

				Tiefensuche		Random	
Programm	E	#CZ	symZ	#Z	Zeit	#Z	Zeit
Addieren	S	240	Array($\leq$ 10)	20	9.82	20	9.82
Branch	S	98	Integer($\geq$ 8)	4	6.24	4	6.21
BubbleSort	S	274	Array($\leq$ 7)	11826	6253.85	11826	7559.86
FibRek	S	195	Integer($\leq$ 15)	30	186.93	30	187.83
Loop	S	106	Loopcounter($\leq$ 450)	902	2681.5	902	2446.65
Max	S	256	Array($\leq$ 15)	65534	69043.90	65534	72335.20
MergeSort	S	461	Array($=$ 7)	80640	30277.50	80640	26003.90
QuickSort	S	311	Array($\leq$ 7)	77052	31634.20	77052	33086.90
Sets	TO	536	Arrays($\leq$ 8)	TO	TO	TO	TO
StringToNum	S	277	char*($\leq$ 10)	114	10.74	114	10.70

Legende:

E – Ergebnis

S – safe (Assertion passed)

#CZ – Anzahl Assemblercodezeilen des Eingabeprogrammes

symZ – Was ist symbolisch?

#Z – Anzahl Zustände

TO – Timeout bei $>$ 24 Stunden

Zeit – Laufzeit in s

Tabelle 13: Ergebnisse vom skalierten Bubblesort

				Tiefensuche	
Programm	E	#CZ	symZ	#Z	Zeit
Bubblesort	S	274	Array(≤ 3)	18	7.69
Bubblesort	S	274	Array(≤ 5)	306	62.02
Bubblesort	S	274	Array(≤ 7)	11826	8894.70
Bubblesort	TO	274	Array(≤ 10)	TO	TO

Legende:

E – Ergebnis

S – safe (Assertion passed)

#CZ – Anzahl Assemblercodezeilen des Eingabeprogrammes

symZ – Was ist symbolisch?

#Z – Anzahl Zustände

TO – Timeout bei $>$ 24 Stunden

Zeit – Laufzeit in s

6 ZUSAMMENFASSUNG UND AUSBLICK

In dieser Arbeit wurde die symbolische Ausführung eines RISC-V-Prozessors vorgestellt, genannt Sym-ISS. Als Ausgangspunkt wurde auf einem in der Arbeitsgruppe Rechnerarchitektur entwickelten RISC-V ISS aufgesetzt.

Mit Hilfe des entwickelten Sym-ISS können Programme nicht mehr nur konkret, sondern symbolisch ausgeführt werden. Dafür wurden symbolische Variablen, Assertions und Assumptions eingeführt, die in den Eingabeprogrammen definiert und durch den Sym-ISS verarbeitet werden können. Symbolische Variablen haben einen symbolischen Speicher zur Folge.

Im nächsten Schritt wurden alle Instruktionsausführungen angepasst, so dass Operationen auf den symbolischen Variablen ausgeführt werden können. Jede Programmzeile im Eingabeprogramm bildet mindestens einen Zustand. Ein Zustand enthält jeweils die Pfadbedingung, den program counter und den Speicher. Stößt der Sym-ISS bei der Abarbeitung eines Eingabeprogrammes auf eine Verzweigung, so wird der aktuelle Zustand geclont. Es gibt daher einen Zustand für den then-Teil und einen für den else-Teil.

Diese unterschiedlichen Zustände werden durch einen Scheduler verwaltet. Hierfür gibt es zwei unterschiedliche Explorationsstrategien, die entscheiden, welcher Zustand bei einem Branching als Nächstes ausgewählt wird. Dafür wurde einmal die Tiefensuche gewählt, die in bisherigen Ausführungs-Engines verwendet wurde und des Weiteren eine zufällige Variante.

Diese beiden Strategien wurden innerhalb der Evaluation in Verbindung mit verschiedenen Eingabeprogrammen untersucht. Jedes Eingabeprogramm wird einmal fehlerhaft und einmal fehlerfrei getestet. Eine gewonnene Erkenntnis ist, dass die Tiefensuche je nach Struktur des Eingabeprogrammes zu einer sehr langen Laufzeit führen kann. Die zufällige Auswahl eines Zustandes ist daher häufig besser geeignet bei der Wahl des nächsten zu explorierenden Zustandes.

Durch die vorliegende Arbeit konnte herausgefunden werden, dass eine symbolische Ausführung möglich ist. Sym-ISS findet in den hier evaluierten Programmen alle eingebauten Fehler und erreicht durch eine zufällige Auswahl des nächsten Zustandes bei einem Branch eine gute Laufzeit.

Basierend auf dieser Arbeit ergeben sich zahlreiche weitere Forschungsfragen, z.B. Optimierungen hinsichtlich der Explorationstrategien von Zuständen. Vielversprechend sind beispielsweise die von Cadar u. a. (2008) vorgeschlagenen drei Explorationsstrategien: Diese sind die Anzahl der bisherigen Sprünge, die Anzahl der durch diesen Zustand entdeckten bisher unbehandelten Programmzeilen und die Distanz zur nächsten bisher unbehandelten Programmzeile. Die

genannten Kriterien zielen darauf ab, dass das Programm nicht in einem Pfad fest steckt, sondern Zustände von verschiedenen Pfaden ausgewählt werden. Des Weiteren sollen sie die Gesamteffektivität des Programms verbessern.

Die erste Strategie bestimmt für alle drei Kriterien die Maxima und ihre entsprechenden Zustände. Es stehen nun drei Zustände zur Auswahl, von denen einer zufällig ausgewählt wird. Die zweite Explorationsstrategie berechnet für jeden Zustand die Summe von allen drei Kriterien. Davon wird der Zustand gewählt, der die höchste Summe aufweist. Die dritte Strategie wählt einen zufälligen Zustand aus, wobei die Wahrscheinlichkeit proportional zur Summe der drei Kriterien ist.

Eine weitere Möglichkeit wäre die Unterstützung von Systemaufrufen. Auch weitere Hardware-Komponenten wie Busse oder Sensorik bieten großes Potential und Herausforderungen, insbesondere im Hinblick auf die Modellierung geeigneter Abstraktionen sowie der effizienten Umsetzung im Kern des ISS selbst. Dabei kommt der Behandlung von Interrupts eine zentrale Rolle zu.

Symbolische Ausführung von Multithread-Programmen des RISC-V-Prozessors könnte ebenfalls umgesetzt werden.

A LITERATURVERZEICHNIS

[Asanović u. Patterson 2014] Asanović, K. ; Patterson, D.: *The Case for Open Instruction Sets*. https://www.linleygroup.com/mpr/login.php?session_id=qbakb9vslnl78iqlvtasjnnua6&return_url=/mpr/article.php?id=11267&num=5210. Version: 2014

[Baldoni u. a. 2018] Baldoni, R. ; Coppa, E. ; D'Elia, D. C. ; Demetrescu, C. ; Finocchi, I.: *A Survey of Symbolic Execution Techniques*. https://arxiv.org/pdf/1610.00502.pdf. Version: 2018

[Bjørner u. de Moura 2008] Bjørner, N. ; Moura, L. de: *Z3: An Efficient SMT Solver*. Springer-Verlag, 2008

[Bjørner u. de Moura 2014] Bjørner, N. ; Moura, L. de: *Applications of SMT solvers to Program Verification*. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/nbjorner-smt-application-chapter.pdf. Version: 2014

[Cadar u. a. 2008] Cadar, C. ; Dunbar, D. ; Engler, D.: *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. https://llvm.org/pubs/2008-12-OSDI-KLEE.pdf. Version: 2008

[Cadar u. Koushik 2013] Cadar, C. ; Koushik, S: *Symbolic execution for Software Testing: Three Decades Later*. http://delivery.acm.org/10.1145/2410000/2408795/p82-cadar.pdf?ip=134.102.168.44&id=2408795&acc=PUBLIC&key=2BA2C432AB83DA15%2E9428DC60D3406798%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1535532064_9336dd78d487b36aa150882ba5bc5e8f. Version: 2013

[Dutertre u. de Moura 2006] Dutertre, B. ; Moura, L. de: *A Fast Linear-Arithmetic Solver for DPLL(T)**. http://leodemoura.github.io/files/cav06.pdf. Version: 2006

[Herdt u. a. 2018] Herdt, Vladimir ; Grosse, Daniel ; Le, Hoang M. ; Drechsler, Rolf: Extensible and Configurable RISC-V based Virtual Prototype. In: *Forum on Specification and Design Languages*, 2018

[King 1976] King, J. C.: *Symbolic Execution and Program Testing*. http://delivery.acm.org/10.1145/370000/360252/p385-king.pdf?ip=134.102.175.25&id=360252&acc=ACTIVE%20SERVICE&key=2BA2C432AB83DA15%2E9428DC60D3406798%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1533034889_58dcfa31072be504a7f8cdad000de0bf. Version: 1976

[Kroening u. Strichman 2008] Kroening, D. ; Strichman, O.: *Decision Procedures*. Springer-Verlag, 2008

[Limaye u. Seshia 2010] Limaye, R. S. ; Seshia, S. A.: *Electrical Engineering and Computer Sciences*. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-67.pdf. Version: 2010

[Oed u. Eckstein 2018] Oed, R. ; Eckstein, M.: *RISC-V: So funktioniert die offene Befehlssatzarchitektur*. https://www.elektronikpraxis.vogel.de/risc-v-so-funktioniert-die-offene-befehlssatzarchitektur-a-746394/?cmp=sm-tw-swyn&utm_source=twitter&utm_medium=sm&utm_campaign=twitter-swyn. Version: 2018

[RISC-V Foundation 2015] RISC-V Foundation: *About the RISC-V ISA*. https://riscv.org/risc-v-isa/. Version: 2015

[The SMT-LIB Initiative 2003] The SMT-LIB Initiative: *SMT-LIB*. http://smtlib.cs.uiowa.edu/. Version: 2003

[Waterman u. Asanovic̀ 2017] Waterman, A. ; Asanovic̀, K.: *The RISC-V Instruction Set Manual*. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. Version: 2017

B ABBILDUNGSVERZEICHNIS

C TABELLENVERZEICHNIS

D VOLLSTÄNDIGE AUFLISTUNG ALLER INSTRUKTIONEN

In diesem Abschnitt werden alle Instuktionen der ISA aufgelistet. Es werden dazu alle SMT-Ausdrücke für jede Instruktion dargestellt.

Tabelle 14: Abbildung der Instruktionen auf SMT-Ausdrücke

Name	Operation	Syntax	SMT-Ausdruck
LUI	$\$r_d$ = (imm $\ll$ 12)	lui $\$r_d$, imm	$\phi(\$r_d)$ = $imm \ll 12$
AUIPC	$\$r_d$ = pc + (imm $\ll$ 12)	auipc $\$r_d$, imm	$\phi(\$r_d)$ = pc + (imm $\ll$ 12)
JAL	$\$r_d$ = pc + 4; pc = pc + imm	jal label	$\phi(\$r_d)$ = pc $\wedge$4
JALR	$\$r_d$ = $\$r_s$	jalr $\$r_d$, $\$r_s$	-
BEQ	if $\$r_{s1}$ ==$\$r_{s2}$ pc=imm	beq $\$r_{s1}$, $\$r_{s2}$, imm	-
BNE	if $\$r_{s1}$!=$\$r_{s2}$ pc=imm	bne $\$r_{s1}$, $\$r_{s2}$, imm	-
BLT	if $\$r_{s1}$ < $\$r_{s2}$ pc=imm	blt $\$r_{s1}$, $\$r_{s2}$, imm	-
BGE	$\$r_d$ = $\$r_{s1}$ + $\$r_{s2}$	bge $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	-
BLTU	$\$r_d$ = $\$r_{s1}$ + $\$r_{s2}$	bltu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	-
BGEU	$\$r_d$ = $\$r_{s1}$ + $\$r_{s2}$	bgeu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	-
LB	$\$r_d$ = mem[$\$r_{s1}$ + imm]	lb $\$r_d$, imm($\$r_{s1}$)	$\phi(\$r_d)$=$\phi$(mem[$\$r_{s1}$ + imm])
LH	$\$r_d$ = mem[$\$r_{s1}$ + imm ... $\$r_{s1}$ + imm+1]	lh $\$r_d$, imm($\$r_{s1}$)	$\phi(\$r_d)$=$\phi$(mem[$\$r_{s1}$ + imm ... $\$r_{s1} + imm + 1$])
LW	$\$r_d$ = mem[$\$r_{s1}$ + imm ... $\$r_{s1} + imm + 3$]	lw $\$r_d$, imm($\$r_{s1}$)	$\phi(\$r_d)$=$\phi$(mem[$\$r_{s1}$ + imm ... $\$r_{s1} + imm + 3$])
LBU	$\$r_d$ = mem[$\$r_{s1}$ + imm] & #xFF	lbu $\$r_d$, imm($\$r_{s1}$)	$\phi\$r_d$ = ϕ(mem[$\$r_{s1}$ + imm]) & #xFF
LHU	$\$r_d$ = mem[$\$r_{s1}$ + imm...$\$r_{s1}$ + imm+1] & #xFFFF	lhu $\$r_d$, imm($\$r_{s1}$)	$\phi(\$r_d)$ = ϕ(mem[$\$r_{s1}$ + imm...$\$r_{s1}$ + imm+1]) & #xFFFF
SB	mem[$\$r_{s1}$ + imm] = $\$r_{s2}$ & #xFF	sb $\$r_{s2}$, imm($\$r_{s1}$)	ϕ(mem[$\$r_{s1}$ + imm]) = $\phi(\$r_{s2})$ & #xFF
SH	mem[$\$r_{s1}$ + imm ... $\$r_{s1}$ + imm + 1] = $\$r_{s2}$ & #xFFFF	sh $\$r_{s2}$, imm($\$r_{s1}$)	ϕ(mem[$\$r_{s1}$ + imm ... $\$r_{s1}$ + imm + 1]) = $\phi(\$r_{s2})$ & #xFFFF
SW	mem[$\$r_{s1}$ + imm ... $\$r_{s1}$ + imm + 3] = $\$r_{s2}$	sw $\$r_{s2}$, imm($\$r_{s1}$)	ϕ(mem[$\$r_{s1}$ + imm ... $\$r_{s1}$ + imm + 3]) = $\phi(\$r_{s2})$
ADDI	$\$r_d$ = $\$r_{s1}$ + imm	addi $\$r_d$, $\$r_{s1}$, imm	$\phi(\$r_d)$=$\phi(\$r_{s1})$+ imm
SLTI	if $\$r_{s1}$ < imm then $\$r_d$ = 1 else $\$r_d$ = 0	slti $\$r_d$, $\$r_{s1}$, imm	$\phi(r_d)$ = ite($\phi(\$r_{s1}$ < imm),#x1,#x0)
SLTIU	if $\$r_{s1} \overset{U}{<}$ imm then $\$r_d$ = 1 else $\$r_d$ = 0	sltiu $\$r_d$, $\$r_{s1}$, imm	$\phi(r_d)$ = ite($\phi(\$r_{s1} \overset{U}{<}$ imm),#x1,#x0)
XORI	$\$r_d$ = $\$r_{s1}$ $\wedge$ imm	xori $\$r_d$, $\$r_{s1}$, imm	$\phi(\$r_d)$ = $\phi(\$r_{s1})$ $\wedge$ imm
ORI	$\$r_d$ = $\$r_{s1}$ | imm	ori $\$r_d$, $\$r_{s1}$, imm	$\phi(\$r_d)$=$\phi(\$r_{s1})$ | imm

Tabelle 14: Abbildung der Instruktionen auf SMT-Ausdrücke

Beschreibung	Operation	Syntax	SMT-Ausdruck
ANDI	$\$r_d = \$r_{s1}$ & imm	andi $\$r_d$, $\$r_{s1}$, imm	$\phi(\$r_d)$=$\phi(\$r_{s1})$& imm
SLLI	$\$r_d = \$r_{s1} \ll$ shamt	slli $\$r_d$, $\$r_{s1}$, shamt	$\phi(\$r_d) = \phi(\$r_{s1}) \ll$ shamt
SRAI	$\$r_d = \$r_{s1} \gg$ shamt	srai $\$r_d$, $\$r_{s1}$, shamt	$\phi(\$r_d)$= $\phi(\$r_{s1}) \gg$ shamt
ADD	$\$r_d = \$r_{s1} + \$r_{s2}$	add $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} + \$r_{s2})$
SUB	$\$r_d = \$r_{s1} - \$r_{s2}$	sub $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} - \$r_{s2})$
SLL	$\$r_d = \$r_{s1} \lll \$r_{s2}$	sll $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} \lll \$r_{s2})$
SLT	if $\$r_{s1} < \$r_{s2}$ then $\$r_d = 1$ else $\$r_d = 0$	slt $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d)$ =ite($\phi(\$r_{s1}) < \$r_{s2}$),#$x$0,#$x$1)
SLTU	if $\$r_{s1} \overset{U}{<} \$r_{s2}$ then $\$r_d = 1$ else $\$r_d = 0$	sltu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d)$ =ite($\phi(\$r_{s1}) \overset{U}{<} \$r_{s2}$),#$x$0,#$x$1)
XOR	$\$r_d = \$r_{s1} \wedge \$r_{s2}$	xor $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \$r_{s1} \wedge \$r_{s2}$
SRL	$\$r_d = \$r_{s1} \ggg \$r_{s2}$	srl $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d)$ =$\phi(\$r_{s1} \ggg \$r_{s2})$
SRA	$\$r_d = \$r_{s1} \gg \$r_{s2}$	sra $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d)$ =$\phi(\$r_{s1} \gg \$r_{s2})$
OR	$\$r_d = \$r_{s1} \mid \$r_{s2}$	ori $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d)$=$\phi(\$r_{s1} \mid \$r_{s1})$
AND	$\$r_d = \$r_{s1}$ & $\$r_{s2}$	and $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d)$=$\$r_{s1}$& $\$r_{s2}$
ECALL	umgebungsabhängig	ecall	-
EBREAK	umgebungsabhängig	ebreak	-
MUL	$\$r_d = \$r_{s1} * \$r_{s2}$	mul $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} * \$r_{s2})$
MULH	$\$r_d = (\$r_{s1} * \$r_{s2}) \gg 32$	mulh $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} * \$r_{s2}) \gg 32$
MULHSU	$\$r_d = (\$r_{s1} \overset{SU}{*} \$r_{s2}) \gg 32$	mulhsu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} \overset{SU}{*} \$r_{s2}) \gg 32$
MULHU	$\$r_d = (\$r_{s1} \overset{UU}{*} \$r_{s2}) \gg 32$	mulhu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} \overset{UU}{*} \$r_{s2}) \gg 32$
DIV	$\$r_d = (\$r_{s1} / \$r_{s2})$	div $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} / \$r_{s2})$
DIVU	$\$r_d = \$r_{s1} \overset{U}{/} \$r_{s2}$	divu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} \overset{U}{/} \$r_{s2})$
REM	$\$r_d = \$r_{s1}$ % $\$r_{s2}$	rem $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1}$ % $\$r_{s2})$
REMU	$\$r_d = \$r_{s1} \overset{U}{\%} \$r_{s2}$	remu $\$r_d$, $\$r_{s1}$, $\$r_{s2}$	$\phi(\$r_d) = \phi(\$r_{s1} \overset{U}{\%} \$r_{s2})$