Institut für theoretische Informatik

Leibniz Universität Hannover

Bachelorarbeit

Synthesis of Machine Programs from Execution Traces

von Yannik Mahlau
Matr.Nr. 10013886

Juli 2020

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Maurice Chandoo

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel und Quellen als angegeben verwendet habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Yannik Mahlau

Contents

1. Introduction

Novice programmers may have difficulties learning how to program using a specific programming language. The reason is most programming languages require one to use complex concepts, even if the programmer only desires to write a simple algorithm. Examples for such concepts are *pointer* in C or *monads* in Haskell. The former is part of any data structure in C programs; the latter is required for program input and output in Haskell programs. This overhead in complexity makes learning difficult. At first glance there are not many alternatives to this learning approach. The learner has to receive feedback to become aware of possible mistakes. Implementing an algorithm may be difficult, but it allows for testing and receiving immediate feedback. To solve this dilemma, Chandoo created a method which supports testing algorithms without implementing them [Cha20a]. The learner plays the algorithm as a game. They perform actions on buttons provided by a machine representation. The information required to decide what action to perform is given by the predicates of the machine. One may imagine a predicate as an indicator light, which presents partial information about the current state of the machine. The user generates a trace of the program by taking actions and checking predicates. This trace is then used to synthesize a program.

As part of this thesis a command line interface (CLI) and graphical user interface (GUI) was developed. They provide a programming environment to use the described method. The CLI presents functionality for extending traces and programs as well as checking consistency between them. The GUI is built on top of the CLI and presents an additional function for creating traces by playing the game described above.

In the second chapter, I present different machine models used in the framework. The concept of these models is the basis for machine programs and traces, which are explained in the third chapter. With these three concepts, it is possible to generate traces by playing the game mentioned above to synthesize a machine program from traces. This process is described in Section 3.4. The fourth chapter illustrates the use of the CLI and GUI to automate parts of the process. Additionally, the complete process of developing a machine program for multiplication using the GUI is demonstrated in Section 4.3. Lastly, the details of the CLI and GUI implementation are presented in order to facilitate further development.

2. Models of Computation

Models of Computation (MoC) are a special kind of notional machine. Notional machines are general abstractions of a machine with the purpose of explaining its behaviour [Sor13]. An MoC consists of machine states, operations, and predicates. One may think of a machine state as numbers in registers, bits in memory, or items on a stack. Operations are actions that alter the machine state. Therefore, an operation is a transition from one valid machine state into another. Predicates are binary indicators, which display partial or full information about the machine state to the user.

Definitions 2.1, 2.2, and 2.3 originate in the work of Chandoo [Cha20b]. Definition 2.4 is a formalization of the Turing Machine used in the framework [Cha20a].

Mathematically, an MoC has the following attributes. The term $\{X \to Y\}$ denotes the set of all functions from X to Y.

Definition 2.1 *A Model of Computation (MoC) is a triple (S, O, P), where*

$$S \text{ is a countable set}$$
$$O = \left\{S \to S\right\} \text{ is finite}$$
$$P = \left\{S \to \{0, 1\}\right\} \text{ is finite}$$

S has to be countable to ensure that its elements can be represented as a string. Functions $o \in O$ and $p \in P$ do not need to be computable in the general definition; however, it only makes sense to use computable operations and predicates in the framework, because otherwise it would not be possible to create traces.

2.1. Counter Machines

In the MoC of a Counter Machine (CM), the machine state has $k \in \mathbb{N}$ registers, each holding one integer value $x_i \in \mathbb{N}_0$ for $1 \leq i \leq k$. The only possible operations are incrementing or decrementing a single register by one or doing nothing. A CM has k predicates, which indicate if the content of a specific register is equal to zero. The concept of counter machines is based on the work of Minsky [Min67] and Lambek [Lam61].

Definition 2.2 *A Counter Machine (CM) with k registers is an MoC (S,O,P) with:*

$$S := \mathbb{N}_0^k$$
$$O := \{\texttt{`Ri+1'}, \texttt{`Ri-1'} \mid 1 \leq i \leq k\} \cup \{\texttt{`NOP'}\}$$
$$P := \{\texttt{`Ri=0'} \mid 1 \leq i \leq k\}$$

The operations `'Ri+1'`, `'Ri-1'` *and* `'NOP'` *(No Operation) are defined as:*

$$\begin{aligned}
\texttt{'Ri+1'} &:= \Big(x_1, \ldots, x_k\Big) \mapsto \Big(x_1, \ldots, x_{i-1}, x_i + 1, x_{i+1}, \ldots, x_k\Big) \\
\texttt{'Ri-1'} &:= \Big(x_1, \ldots, x_k\Big) \mapsto \Big(x_1, \ldots, x_{i-1}, x_i - 1, x_{i+1}, \ldots, x_k\Big) \\
\texttt{'NOP'} &:= \Big(x_1, \ldots, x_k\Big) \mapsto \Big(x_1, \ldots, x_k\Big)
\end{aligned}$$

The predicates `'Ri=0'` *are defined as:*

$$\texttt{'Ri=0'} := \Big(x_1, \ldots, x_k\Big) \mapsto 1 \Leftrightarrow x_i = 0$$

In the Haskell framework, a machine state is represented as a list of integers [Cha20a].

2.2. Stack Machines

In the MoC of a Stack Machine (SM), a machine state has $k \in \mathbb{N}$ registers, each representing one stack. Every stack contains several characters from a given input alphabet as items. Possible operations are pushing a new item on the stack, removing the top item from the stack, or doing nothing. The predicates indicate either if the stack is empty or if the top item on the stack is equal to a given character of the input alphabet. The input alphabet has to be a subset of the legal input alphabet specified in the framework [Cha20a]. Legal input characters are:

- Upper and lower case letters: $a, \ldots, z, A, \ldots, Z$
- Digits: $0, \ldots, 9$
- Symbols: `+ - * / = _ .  , # $ % & ' " ( ) [ ] { } < >`

A stack machine is defined with the following attributes. The term Σ^* refers to the set of all finite words over symbols from Σ.

Definition 2.3 *A Stack Machine (SM) with k stacks and an input alphabet $\Sigma = \{c_1, \ldots, c_l\}$ is a triple (S, O, P) with:*

$$\begin{aligned}
S &:= \Big(\Sigma^*\Big)^k \\
O &:= \{\, \texttt{'Ri+c'}, \texttt{'Ri-'} \mid 1 \leq i \leq k,\, c \in \Sigma\} \cup \{\, \texttt{'NOP'}\} \\
P &:= \{\, \texttt{'Ri=c'}, \texttt{'Ri='} \mid 1 \leq i \leq k,\, c \in \Sigma\}
\end{aligned}$$

The operations `'Ri+c'`, `'Ri-'` *and* `'NOP'` *are defined as:*

$$\begin{aligned}
\texttt{'Ri+c'} &:= \Big(x_1, \ldots, x_k\Big) \mapsto \Big(x_1, \ldots, x_{i-1}, x_i \oplus c, x_{i+1}, \ldots, x_k\Big) \\
\texttt{'Ri-'} &:= \Big(x_1, \ldots, x_k\Big) \mapsto \Big(x_1, \ldots, x_{i-1}, x_i\prime, x_{i+1}, \ldots, x_k\Big) \\
\texttt{'NOP'} &:= \Big(x_1, \ldots, x_k\Big) \mapsto \Big(x_1, \ldots, x_k\Big)
\end{aligned}$$

where $x_i \in \Sigma^*$ for $1 \leq i \leq k$ and $c \in \Sigma$. The operator $\oplus$ refers to the concatenation of two strings. x_i' is the word x_i without its last symbol, or x_i, if x_i is empty. The predicates are defined as:

$$\texttt{'Ri='} := \Big(x_1, \ldots, x_k\Big) \mapsto 1 \Leftrightarrow |x_i| = 0$$
$$\texttt{'Ri+c'} := \Big(x_1, \ldots, x_k\Big) \mapsto 1 \Leftrightarrow \textit{Last symbol of } x_i \textit{ is } c$$

In the Haskell framework, the machine state of a stack machine is represented as a list of strings [Cha20a]. Each string represents one stack.

2.3. Generic Models

All models other than counter and stack machines are represented as generic models. They do not have specific constraints for machine states, operations or predicates. Thus, generic models allow developers to implement other MoCs. Currently, the only generic MoC implemented is a Turing Machine (TM).

The machine state of a TM is a tape with cells. It also has an integer representing the current position on the tape. Each cell contains a character from a specified input alphabet Σ. The default character in each cell is a designated blank symbol $a \in \Sigma$ specified by the user. The tape is arbitrarily long in both directions, but the machine state only represents the visited cells. All non-visited cells contain the blank symbol and therefore do not need to be represented. The representation of a machine state is:

$$\Big((c_0, \ldots, c_k), i\Big)$$

with $c_0, \ldots, c_k \in \Sigma$, $0 \leq i \leq k$ and $i, k \in \mathbb{N}_0$. The integer i represents the index of the current cell position. The operations of a Turing machine are moving one cell to the right, moving one cell to the left, or writing a character $c \in \Sigma$ to the current cell. A new cell is prepended (appended) to the representation of the machine state if the current position is the leftmost (rightmost) visited cell and one moves the current position to the left (right). This new cell always contains the blank character. A Turing machine has one predicate for every character $c \in \Sigma$. The predicates indicate if the current cell contains the given character [Cha20a].

Definition 2.4 *A Turing Machine (TM) with an input alphabet Σ and a designated blank character $a \in \Sigma$ is a triple (S, O, P) with:*

$$S := \{(w, i) \mid w \in \Sigma^+,\ i \in \mathbb{N}_0,\ 0 \leq i \leq |w|\}$$
$$O := \{\ \texttt{'Wx'} \mid x \in \Sigma\} \cup \{\ \texttt{'L'}, \texttt{'R'}\}$$
$$P := \{\ \texttt{'=x'} \mid x \in \Sigma\}$$

The term Σ^+ refers to the set of all finite, non-empty words of symbols from Σ. The

operations **‘Wx’**, **‘L’** *and* **‘R’** *are defined as:*

$$\text{‘Wx’} := \Big((c_0, \ldots, c_k), i\Big) \mapsto \Big((c_0, \ldots, c_{i-1}, x, c_{i+1}, \ldots, c_k), i\Big)$$

$$\text{‘L’} := \Big((c_0, \ldots, c_k), i\Big) \mapsto \begin{cases} \Big((c_0, \ldots, c_k), i-1\Big) & i \neq 0 \\ \Big((a, c_0, \ldots, c_k), 0\Big) & \textit{otherwise} \end{cases}$$

$$\text{‘R’} := \Big((c_0, \ldots, c_k), i\Big) \mapsto \begin{cases} \Big((c_0, \ldots, c_k), i+1\Big) & i \neq k \\ \Big((c_0, \ldots, c_k, a), k+1\Big) & \textit{otherwise} \end{cases}$$

The predicate **‘=x’** *is defined as:*

$$\text{‘=x’} := \Big((c_0, \ldots, c_k), i\Big) \mapsto 1 \Leftrightarrow c_i = x$$

In the Haskell framework, a machine state of a TM is represented as a String which contains the tape content as an escaped string and the tape position as a decimal value [Cha20a].

3. Trace-based Programming

Trace-based programming is a systematic approach to programming. It consists of simple and easily executable steps for program development. In contrast to ad-hoc programming, it also provides clear guidelines on how to develop a program. The method is based on the concepts of machine programs and traces. In this chapter these concepts as well as the syntactical rules for writing one's machine programs or traces are presented. Furthermore, the connection between machine programs and traces can be analyzed by using consistency, which is presented in Section 3.3. Lastly, the steps of generating a trace and synthesizing a machine program are explained in detail.

Definition 3.1, 3.4, 3.6, and 3.8 were made by Chandoo [Cha20b].

3.1. Machine Programs

Machine programs are always defined with an MoC. One may imagine a machine program as a finite state machine. Every state of the machine program except for the start state is associated with an operation of the MoC. The machine program changes its states based on the information about the machine state given by the predicates.

Definition 3.1 *An M-Program Q is a quadruple (G, v_0, α, β), where $M = (S, O, P)$ is an MoC:*

$$
\begin{aligned}
&G = (V, E) \text{ is a directed Graph}\\
&v_0 \in V \text{ with in-degree 0}\\
&\alpha : V \backslash \{v_0\} \to O\\
&\beta : \Big(E \cup Z\Big) \to \Big\{\{0,1\}^{|P|} \to \{0,1\}\Big\}, \quad Z \subseteq \Big(V \times \{\texttt{End}\}\Big)
\end{aligned}
$$

G is called flow control graph and v_0 is the start state of Q. Each vertex of G is a program state. α defines the operation for each vertex in the control flow graph. β is the edge predicate of G, which defines the transitions between program states. For all program states, only one of the edge predicates of their outgoing edges is allowed to be true for any given machine state. This ensures a deterministic behaviour of the program.

For example, suppose one has two bags of coins. The first bag contains coins with a value of one; the second contains coins with a value of two. The task is to compute the total value of coins in both bags. The problem can be modelled as a counter machine with two registers. The first register contains the number of coins in the first bag, the second the number of coins in the second bag. At the end of the computation, the total

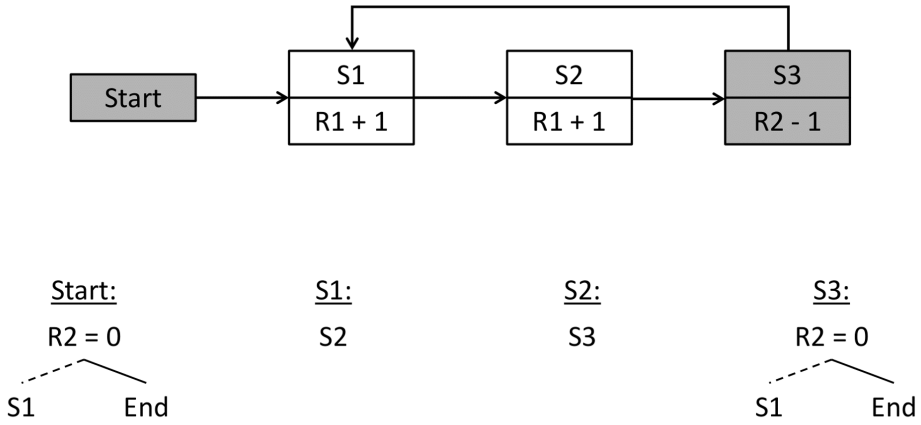

Figure 3.1.: Control Flow Graph for Coin Bag Example

coin value is supposed to be in the first register. Formally, we want to compute the function:

$$(x, y) \mapsto (x + 2y, 0) \text{ with } x, y \in \mathbb{N}_0$$

Figure 3.1 shows the control flow graph of a machine program that solves this task. The vertices "S1", "S2" and "S3" have an operation associated with them. Additionally, vertices "Start" and "S3" are marked gray as they are terminal states where execution may stop. The edge predicate α is defined implicitly by binary decision trees (BDT). In the BDTs, the dotted lines are taken if the predicate is false, the solid lines otherwise.

The control flow graph itself is a redundant display of the program's functionality. The binary decision trees alone can display all information in G. This approach is used in the syntax of a machine program file.

A machine program file contains code that defines the behaviour of a machine program on a specific MoC. They are text files with the file name extension ".mp". The syntax of a machine program file presented in this section was defined by Chandoo [Cha20a]. A machine program file may contain multiple independent programs, but all programs have to work with the same MoC. Therefore, the user has to define the MoC in the first line (only comments and empty lines may precede this definition). The MoC is defined with the following term, where `ID` is the identifier of the desired model:

```
#MOC ID
```

The MoCs presented in chapter 2 have the identifiers:

Counter Machine: `CM N`
Stack Machine: `SM N C`
Turing Machine: `TM C`

where `N` is the number of registers and `C` the input alphabet. The first character in the input alphabet of a Turing Machine is the designated blank symbol. It is recommended to use the underscore character as a blank symbol, as it is rarely used otherwise and provides good visibility. As an example, the definition of a Stack Machine with four registers and binary input alphabet is:

```
#MOC SM 4 01
```

Afterwards, the user may start writing a program. A new program starts with the line:

```
#PROGRAM programName
```

Programs consist of several program states. Each State consists of a name, an operation, and a binary decision tree. The state `Start` is the first state in every program and has no operation. The name and operation of a state are defined in its first line. The state `Start` begins with the line:

```
Start:
```

Every other program state begins with the following line:

```
stateName / operation:
```

Binary decision trees determine the next state in the program's execution. Each inner node of the BDT contains a predicate; the leaf nodes contain either the name of the next program state or the directive `End` to terminate the execution. The children of each node are indented by one tab in a new line, starting at the root node with one tab indentation.

The first child of a node corresponds to the case that the predicate in this individual node is false, the second to the case that the predicate is true. This may seem counterintuitive for advanced programmers because it is the reverse of the general if-then-else structure in most programming languages. However, the programming method is primarily intended to be used by novice programmers, who do not share this bias yet. The reason why this concept may be surprising is the the lack of keywords in machine programs, which mark the order of cases. This indicates that novice programmers will not have issues in advancing from programming with machine programs to other programming languages. On the contrary, concepts that are marginally different from the status quo can even be easier to remember due to the minimal counterintuitiveness effect [Upa10].

It is also possible to add single line comments to machine program code. A comment starts with two forward slashes (`//`).

Machine Program 3.1 is an example program, which solves the coin bag task mentioned above. It shows the structural similarities between a machine program and the control flow graph, i.e. Figure 3.1.

Two more advanced syntactical constructs are subprogram and function calls. They allow the user to write modularized code. However, the framework currently does not support import statements. Therefore, machine program code can be modularized, but

```
 1 #MOC CM 2
 2 #PROGRAM coins
 3 Start:
 4     R2=0
 5         S1
 6         End
 7
 8 S1 / R1+1:
 9     S2
10
11 S2 / R1+1:
12     S3
13
14 S3 / R2-1:
15     R2=0
16         S1
17         End
```

Machine Program 3.1: Coin Bag Example

has to reside in a single machine program file. Both subprogram and function calls can be used as special, user defined operations.

Subprogram calls execute the specified program on the current machine state. After termination of the subprogram, the execution of the super-program is resumed. They can be executed by using the operation `$` followed by the name of the subprogram. E.g. suppose the subprogram `coins` is to be executed in a program state called `S4` of the super-program. The user can achieve this behaviour by adding the line:

```
S4 / $coins:
```

The user may also specify a permutation $\pi \in S_n$ where n is the number of registers. It permutes the registers for the duration of the subprogram execution. One may imagine that the registers are simply renamed. Let x be the first argument of the permutation. Then, the register x is renamed as register one. If y is the second argument of the permutation then register y is renamed as register two. All registers are relabelled accordingly.

For example, suppose one uses an MoC with four registers and wants to call a subroutine named `subprogram`. The program state `S4` using this call as an operation could look like this:

```
S4 / $subprogram 2 3 1 4:
```

The line above describes the mapping:

$$\begin{aligned}\text{Reg. } 2 &\mapsto \text{Reg. } 1\\ \text{Reg. } 3 &\mapsto \text{Reg. } 2\\ \text{Reg. } 1 &\mapsto \text{Reg. } 3\\ \text{Reg. } 4 &\mapsto \text{Reg. } 4\end{aligned}$$

It is important to remember the direction of the mapping because the inverse mapping would produce a different result. The permutation may also be abbreviated if the last elements are either mapped to themselves or their mapping is not important for the computation of the subroutine. I.e. the mapping of register four in the line above could be omitted. The abbreviation only works for the last elements, because otherwise the mapping would be ambiguous. The framework completes abbreviated permutations by mapping the missing elements of the domain to the respectively missing elements of the codomain in order from lowest to highest register.

Function calls can be used similarly to subprogram calls. They are specified as an operation by using two dollar signs (`$$`) as well as the program name. However, the concept of function calls is different from subprogram calls. The difference is that function calls are executed on a separate machine. The user may specify multiple input registers and a single output register to control the input and output of the other machine. The other machine has to use the same MoC as the calling machine, but this is already enforced as both programs have to reside in the same machine program file. The user has to define the number of input registers and the index of the output register in the header line of a program in order for it to be usable as a function.

```
#PROGRAM name N O
```

`N` is the number of input registers and `O` is the index of the output register. Input registers are the registers from index `1` to `N`. The output register may be an input register.

Similarly, the calling program has to define which of its registers should provide the input as well as the output register to which the result is written. Syntactically, the input and output registers of the calling function are defined as an index list of length $\texttt{N} + 1$ with the last number being the index of the output register:

$$\texttt{State / \$\$func } \texttt{I}_1 \ \texttt{I}_2 \ \ldots \ \texttt{I}_N \ \texttt{O:}$$

An example of a machine program using function calls is Machine Program 3.2, wherein the program `calling` calls the function `inc`. The operation `$$inc 3 2` takes the value in register three, increments it by one, and writes the result to register two. The content of register three is not changed by this operation.

Partial machine programs are programs without fully defined behaviour. Section 3.4 presents more details about synthesizing a complete program from a partial program and traces. The user has to specify an undefined state to separate defined and undefined behaviour. The undefined state needs to be named `Undef` and its binary decision tree may only consist of the directive `End` to terminate execution. It is recommended to

```
 1 #MOC CM 4
 2
 3 //increments register one
 4 #PROGRAM inc 1 1
 5 Start:
 6     S1
 7
 8 S1 / R1+1:
 9     End
10
11 #PROGRAM calling
12 Start:
13     S1
14
15 S1 / $$inc 3 2:
16     End
```

Machine Program 3.2: Function Call

assign the operation `NOP` to the undefined state when using the MoC SM or CM to ensure clear separation to other program behaviour. However, the user may also use any other operation. The initial machine state of a program is the machine state $s_0 \in S$ of its MoC at the beginning of execution.

Definition 3.2 *A machine program Q using an MoC $M = (S, O, P)$ is complete, if for all initial machine states $s \in S$ the execution of Q does not enter an undefined program state. On the contrary, a machine program is called partial if there exists an initial machine state $s \in S$ such that the execution of Q enters an undefined program state.*

Definition 3.3 *The binary decision tree (BDT) of a program state is called partial if at least one of its leaf nodes contains the directive to enter the undefined program state. A BDT is called complete if it is not partial.*

Complete programs may have partial BDTs if the predicates leading to the leaf nodes, which contain the directive to enter the undefined state, are never true. These programs are by definition complete, because they never enter the undefined state.

Machine Program 3.3 is an example of a partial program. It has fully undefined behaviour, which is a useful starting point to develop any program.

Partial BDTs usually occur in complete programs, if the programmer thinks more predicates have to be checked than necessary. But, there is a procedure to eliminate partial BDTs from complete programs.

Definition 3.4 *Let Q, Q' be M-Programs for an MoC $M = (S, O, P)$. Q and Q' are called equivalent if their machine states are equal for all steps of the execution on initial machine state s. This has to hold for all initial machine states $s \in S$.*

```
1 #PROGRAM undef
2 Start:
3     Undef
4
5 Undef / NOP:
6     End
```

Machine Program 3.3: Fully Undefined Program

Theorem 3.5 *For every complete program with partial BDTs exists an equivalent complete program with only complete BDTs.*

Proof. Let Q be a complete program with partial BDTs. One can use the following procedure to eliminate an undefined node from a partial BDT.

First delete the undefined node. This does not change the program's behaviour because the undefined node is unreachable due to the program's completeness. However, a syntactically correct BDT must only have inner nodes with exactly two children. The former parent of the undefined node has only one child at this point. Therefore, replace the former parent with its only child node. This does not change the program's behaviour, because a path in the BDT from the root to the parent node would always continue with the sibling node.

The resulting BDT yields the same behaviour and does not include the undefined node. The process can be repeated for every undefined node in every partial BDT. The resulting program Q' is equivalent to Q, because Q is complete and therefore the deleted undefined nodes unreachable. Naturally, the resulting program Q' is also complete. □

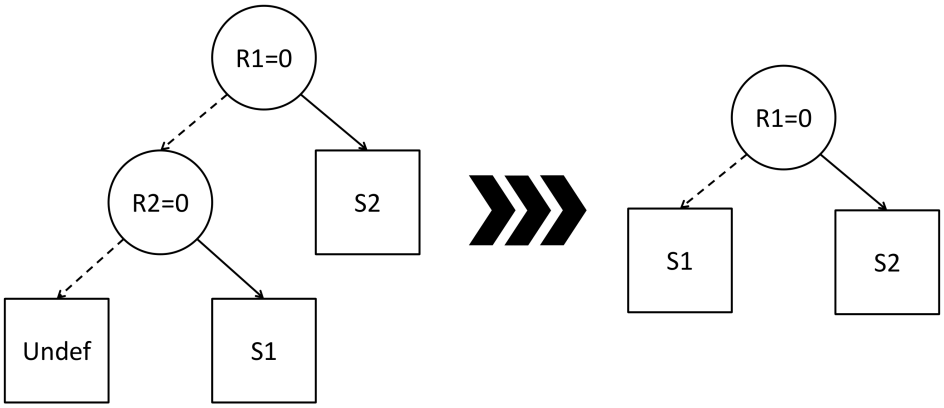

Figure 3.2.: Elimination of Partial BDT in Complete Programs

For example, suppose the program containing the BDT in Figure 3.2 is complete. Again, the dotted lines refer to the case where the predicate is false. First the predicate `R1=0` is checked and then `R2=0`. Because the program is complete, the predicate `R2=0` is always true. Thus, we can replace the subtree beginning at `R2=0` with the node `S1`, which is the sibling of the undefined node.

3.2. Traces

Traces are a documentation of the behaviour of a machine program on a given input. A trace can be represented as a table, wherein every row is one step in the program's execution. The mathematical definition of a trace is based on the state space graph G of the MoC used by the corresponding machine program.

Definition 3.6 *The state space graph $G = (V, E)$ of an MoC (S, O, P) is a directed Graph with:*

$$V := S$$
$$E := \{(s, s') \mid o(s) = s',\ o \in O,\ s, s' \in S\}$$

G represents all possible machine states and the operations changing one machine state to another.

A trace is a path through G, containing the names n and operations o of all vertices along the path. Additionally, the trace stores the machine states s after every operation as well as the sequences of predicates f checked at the corresponding program state. Definition 3.7 is an extension of the definition of traces made by Chandoo [Cha20b].

Definition 3.7 *A trace t through a state space graph $G = (V, E)$ of an MoC $M = (S, O, P)$ used by the M-Program Q is a tuple (n, o, s, f):*

$$n := (\text{'Start'}, n_1, \ldots, n_k) \text{ where } n_1, \ldots, n_k \text{ are any program state names of } Q$$
$$o := (o_1, \ldots, o_k),\ \text{with } o_1, \ldots, o_k \in O \text{ and } o_i(s_{i-1}) = s_i\ \forall 1 \leq i \leq k$$
$$s := (s_0, \ldots, s_k),\ \text{with } s_0, \ldots, s_k \in S$$
$$f := (f_0, \ldots, f_k) \text{ where } f_i = (p_1, \ldots, p_{l_i}),\ p_1, \ldots, p_{l_i} \in P,\ 0 \leq l_i \leq |P|$$

Definition 3.8 *The length $|t|$ of a trace $t = (n, o, s, f)$ is defined as:*

$$|t| = |o|$$

Trace 3.1 is an example of a trace for Program 3.1 of the coin task. The initial machine state is $(3, 2)$, which yields a machine state of $(3 + 2 * 2, 0) = (7, 0)$ at the end of the computation.

Trace files are text files formatted as comma-separated values (CSV). They have the file extension ".tr". Entries of a trace are formatted in a table, in which every line in the file is one row of the trace. The delimiter between entries is a semicolon. A trace file starts with a header line, which has to be

State	Operation	R1	R2	Predicate Sequence
Start		3	2	(R2=0, False)
S1	R1+1	4	2	
S2	R1+1	5	2	
S3	R2-1	5	1	(R2=0, False)
S1	R1+1	6	1	
S2	R1+1	7	1	
S3	R2-1	7	0	(R2=0, True)

Trace 3.1.: Coin Program

```
State;Op.;R1;...;Rn;P.s.
```

for a counter or stack machine. The first column `State` defines the name of the program state in every row. Its first entry has to be the state `Start`.
The second column defines the operation executed in each row. Its first entry is empty, because the program state `Start` does not execute an operation.

The columns `R1` to `R`n represent the machine states of registers one to n in every row. If a machine state is equal to the state in the previous row then its entry may be omitted, which is called sparse representation. Naturally, the user is not allowed to use a sparse machine state representation in the first row of a trace.

The last column defines the predicate sequence checked in every step. Multiple options for formatting predicate sequences are possible. In general, it has to be a string consisting of a predicate, followed by an optional boolean, followed by a predicate, and so on. The first letter of boolean values has to be upper case. A user may add any kind of parentheses for better readability. However, the parentheses have no semantic meaning. The optional boolean values may be omitted because the machine state already contains this information. Notwithstanding, it is recommended to provide all optional boolean values and use normal parentheses for best readability. This is also the output format of the CLI for automatically generated traces. The order of the predicates represents the order of predicates checked in the binary decision tree of a program state. A predicate sequence may also be empty if no predicates are checked. An example of a predicate sequence for a counter machine is:

```
(R1=0, False),(R2=0, True)
```

Generic models have a marginally different trace format. Their header line is defined as:

```
State;Op.;MachineState;P.s.
```

A machine state of a generic model has to be encoded in a single column. This is because generic models may have varying rules for the amount of registers used or they may not use registers at all. All other columns have the function and syntax as described above.

Furthermore, one can use sparse representations for generic machine states or predicate sequences in the same way as described above.

3.3. Consistency

The concept of consistency determines if one or more traces fit to a complete or partial program. This allows a user to check if their traces have the same behaviour as the machine program. The definition of consistency discussed in this section is an extension of the concept of consistency introduced by Chandoo [Cha20b].

Definition 3.9 *Let Q be a machine program and T a set of traces. All $t \in T$ with $t = (n, o, s, f)$ have to fulfil the following conditions in order for T and Q to be consistent. s_0 is the first machine state in s. $t' = (n', o', s', f')$ is the trace generated by the execution of Q with initial machine state s_0. If the last program state n'_l in n' with $l = |t'|$ is not the undefined state, then the required condition is:*

$$t = t'$$

Otherwise, the required conditions are:

$$\begin{aligned} n_i &= n'_i \quad && \forall\, 0 \leq i \leq |n'| - 1 \\ o_i &= o'_i \quad && \forall\, 1 \leq i \leq |o'| - 1 \\ s_i &= s'_i \quad && \forall\, 0 \leq i \leq |s'| - 1 \\ f_i &= f'_i \quad && \forall\, 0 \leq i \leq |f'| - 2 \\ f_k &=_{|f'_k|} f'_k \quad && k = |f'| - 1 \end{aligned}$$

The operation $=_l$ refers to equality up to the length of l. The last predicate sequence f_k with $k = |f'| - 1$ only has to be equal up to the length of f'_k.

Intuitively, all traces have to be equal to the computation of the corresponding machine program up to the point of undefined behaviour. It is only required to check if the last trace row generated by Q describes undefined behaviour, because execution always ends instantly after entering the undefined program state.

However, Definition 3.9 does not imply that a machine program exists, which produces all traces in T. For example, let Q be a partial program consistent and T a set of traces. Then, multiple traces $t \in T$ may describe different behaviour after the point of execution at which Q enters the undefined state. Because of this, synthesis of a machine program Q' from a partial program Q and a set of traces T may fail, even if Q and T are consistent.

3.4. Synthesis of Machine Programs

This section describes the process of developing a complete machine program from a user's perspective using only trace-based programming. The described method was developed by Chandoo [Cha20a]. A precondition for using trace-based programming is that the user has a mental representation of the algorithm they desire to implement. Additionally, all subroutines or function calls used in the algorithm should already be implemented. If they are not already implemented, the user should start by using the method presented in this section to implement the subroutines before proceeding.

The first step is to generate traces of the desired program by playing a machine-computer game. The game can either be played digitally by using the GUI or in an analogue manner by using pen and paper. It consists of two players: machine and computer. The machine simulates a specific MoC with an internal machine state invisible to the computer. The computer may ask the machine if predicates are true for its current machine state or change the machine's internal state with operations. Using these two actions repeatedly, the computer performs the algorithm. A recording of the game is a trace containing operations, machine states, and predicate sequences, but not program states.

Therefore, the second step is the assignment of program states to every row of the generated trace. Two trace rows with different operations cannot correspond to the same program state. However, trace rows with the same operation do not have to correspond to the same program state, because multiple program states may have the same operation. The first program state has to be `Start`.

In the third step, the user has to extend a partial program with the enriched trace. If the user does not have a partial program they should start with a fully undefined program (Machine Program 3.3). By using the information of the trace, the user can add new program states and extend undefined nodes in the binary decision trees.

The user may stop if the machine program is complete after step three. Otherwise, they have to repeat steps one to three with different initial machine states until the program is complete. Any program requires only a finite amount of traces to be complete [Cha20b]. Algorithm 3.1 presents the method in an abbreviated format.

Algorithm 3.1 Synthesis of a Machine Program from Traces

1: $Q \leftarrow$ 'Fully Undefined Program'
2: **while** Q is not complete **do**
3: Choose initial machine state s_0
4: Generate trace t from s_0
5: Assign program states to every row of t
6: Extend Q using t

4. User Manual

Trace-based programming can be tedious if traces have to be written by hand or machine programs have to be manually extended. Therefore, the CLI and GUI were developed to automate these two tasks. This reduces the work of the programmer to finding all critical initial machine states and assigning program states to the rows of every trace generated. This chapter presents the features of CLI as well as GUI and explains how to use them.

4.1. Command Line Interface

The main functions of the command line interface are extending machine programs and traces as well as checking consistency. It uses both machine program and trace files with the syntax described in chapter 3. The CLI can be accessed via any command line tool.

The process of extending a machine program or trace adds the information given by the corresponding trace or machine program. The extension may fail if traces and machine program describe different behaviour. For more information on the algorithms used see Chapter 5.

Machine program and trace files have to be arranged in a specific hierarchical structure to provide a clear connection between a machine program and its corresponding traces. This hierarchical structure requires two layers, because a machine program file may contain multiple programs with each their traces. The top-level contains the machine program file and a folder with the same name as the machine program file, except for the file extension ".mp". This folder contains one subfolder for each program in the machine program file. The subfolders need to have the same name as the program names. Inside the subfolders reside the trace files of the corresponding programs.

For example, a machine program file could contain two programs named `prog1` and `prog2`. The machine program file has the name `machine.mp`. The program `prog1` has two traces, namely `trace1.tr` and `trace2.tr`. Program `prog2` does not have traces. Figure 4.1 depicts the file hierarchy of such composition.

The name of the executable CLI file is `synth_cli.exe`. It may be executed with the command:

```
synth_cli.exe path [-c] [-pname name]
```

The argument `path` is the file path to either a machine program or a trace file. If it is a machine program file, the CLI extends the first program of the file with its corresponding traces given by the file hierarchy. Otherwise, it extends the trace using the corresponding machine program file. If the argument `-c` is provided, the CLI only tests for consistency

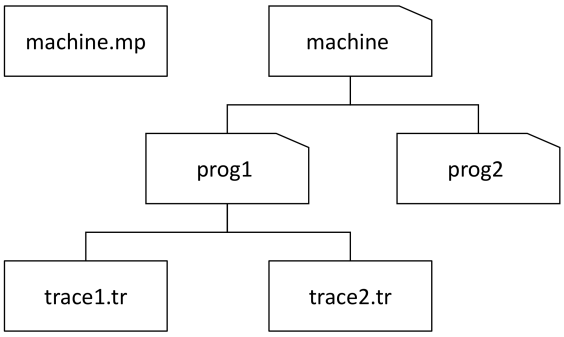

Figure 4.1.: File Hierarchy of Machine Programs and Traces

between the trace and machine program. The user may check the consistency of a single trace by using the path to the trace with the option `-c`. Otherwise, the consistency of the machine program file given by the path with all its traces is checked. The CLI uses the first program in the machine program file by default. Alternatively, the user may specify the option `-pname name`, where `name` is the name of a program in the machine program file. This command instructs the CLI to use the specified program instead of the first program in the file.

For example, suppose one is using the file hierarchy given by Figure 4.1. The command

```
synth_cli.exe machine.mp -c -pname prog1
```

instructs the CLI to check the consistency of machine program `prog1` with all its corresponding traces, namely `trace1.tr` and `trace2.tr`. The option `-pname prog1` is redundant if we assume that `prog1` is the first program in `machine.mp`.

All other functions of the CLI are mainly used by the GUI to access the framework mediated by the CLI. For more information on these other functions see Section 5.2.

4.2. Graphical User Interface

The graphical user interface enhances the functions provided by the CLI with better visuals and ease of use. Additionally, it adds the function of generating traces by playing a simulation of the machine-computer game presented in Section 3.4. It also hides the file hierarchy from the user. The user can only work on a single machine program file per instance of the GUI. Working with multiple machine program files at the same time requires multiple GUI-Windows. To use the graphical user interface, the user needs to have Java version 14.0.1 and OpenJavaFX 14.0.1 or newer installed.

Figure 4.2 displays the main stage of the GUI. It consists of three basic areas. The top left side contains the content of the machine program file. A user may select one program in the file in a drop down menu. By default, the first program in the file is selected. The top right area displays the traces of the selected program in multiple tabs.

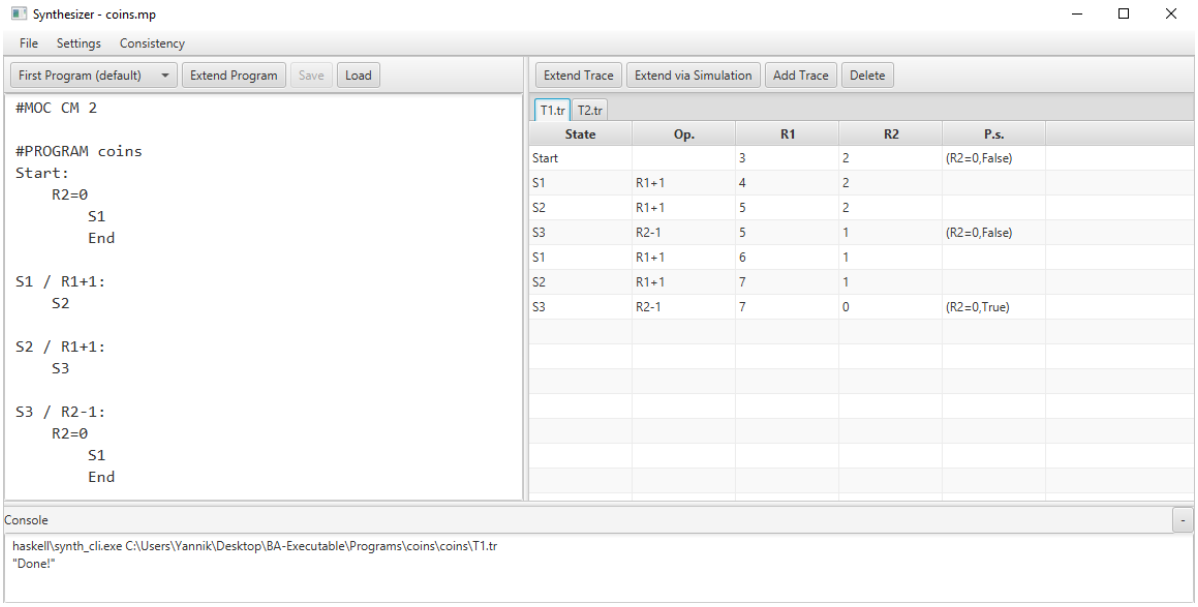

Figure 4.2.: Main Stage of the GUI

Each Tab corresponds to exactly one trace. Tabs are ordered alphabetically. Traces are displayed in a table granted they are formatted correctly. The area at the bottom is a read-only console displaying the output or error messages of the CLI.

The three areas are divided by moveable separators. The user can adjust the width and height of the areas to fit their need. Additionally, the console is collapsible and extendible with the minus and respectively plus button at the right side.

Above the area displaying machine programs and traces reside buttons to alter the respective content. These functions include:

- *Extend Machine Program:* Takes all traces in the trace area and extends the machine program with their information.
- *Save Machine Program:* Saves the text in the program area to the machine program file. If no machine program file exists yet, the user is presented with a prompt to specify a file location. An alternative way to access this function is pressing `CTRL+S` while editing the machine program.
- *Load:* Loads a new machine program file with its traces. If there is unsaved content in the current machine program or traces, the user is asked if they want to save.
- *Extend Trace:* Executes the machine program with the initial machine state of the currently selected trace. If this yields new information, new rows are added to the trace. If the execution of the machine program ended in the undefined state, then it is reported to the user by a message in the console.
- *Extend Manual:* Starts a simulation of the machine computer game, beginning at the last row of the currently selected trace. The trace rows generated by playing

the game are added to the end of the trace. More information can be found below.

- *Add Trace:* Opens a window allowing the user to specify the initial machine state and the name of a new trace. The name has to be unique for all traces of the currently selected program (see file hierarchy). Also, the user may choose to use random values for the initial machine state.
- *Delete:* Deletes the currently selected trace.

Some additional functions are located in the menu bar at the very top of the window. These functions include:

- *Save Trace / Save all Traces:* Saves the content of the currently selected trace or all traces to their trace files.
- *Consistency Current / All Traces:* Checks the consistency between the currently selected trace or all traces with the machine program. The result is displayed in the console.

The functions for extending the machine program, traces, or checking consistency automatically save the content of the machine program and all traces. Additionally, if there is unsaved content when closing the window then the user is asked if they want to save.

The GUI does not contain any machine programs, traces or console messages at startup. The typical workflow either starts with writing a new program, i.e. the fully undefined machine program, or loading an old program file. As soon as the new file is saved or the old one is loaded, the user may start adding and editing traces.

The machine-computer game facilitates the process of writing traces. After clicking the "Extend-Manual" Button, a simulation of the game begins with the last row of the currently selected trace. Figure 4.3 shows this window for a counter machine with four registers. The user is presented with a matrix of buttons on the left side. These buttons execute operations or predicate checks. On the right side the operations and predicate sequences of the trace are displayed.

Clicking the right mouse button while editing the machine program or a trace cell opens a menu with options for text editing, e.g. `Copy` or `Paste`. Alternatively, one may use the following shortcuts:

- `CTRL+C` Copy the selected text.
- `CTRL+V` Paste the copied text.
- `CTRL+Z` Undo the last text edit.
- `CTRL+Y` Redo the last text edit.

Additionally, it is possible to use the shortcut `CTRL+S` to save the machine program.

There are also keyboard shortcuts make it more comfortable to navigate the trace tables. A user may start editing a cell by double-clicking it with the mouse or pressing the `RETURN` Key. While editing a cell, the user may press one of the following key combinations to navigate through the table:

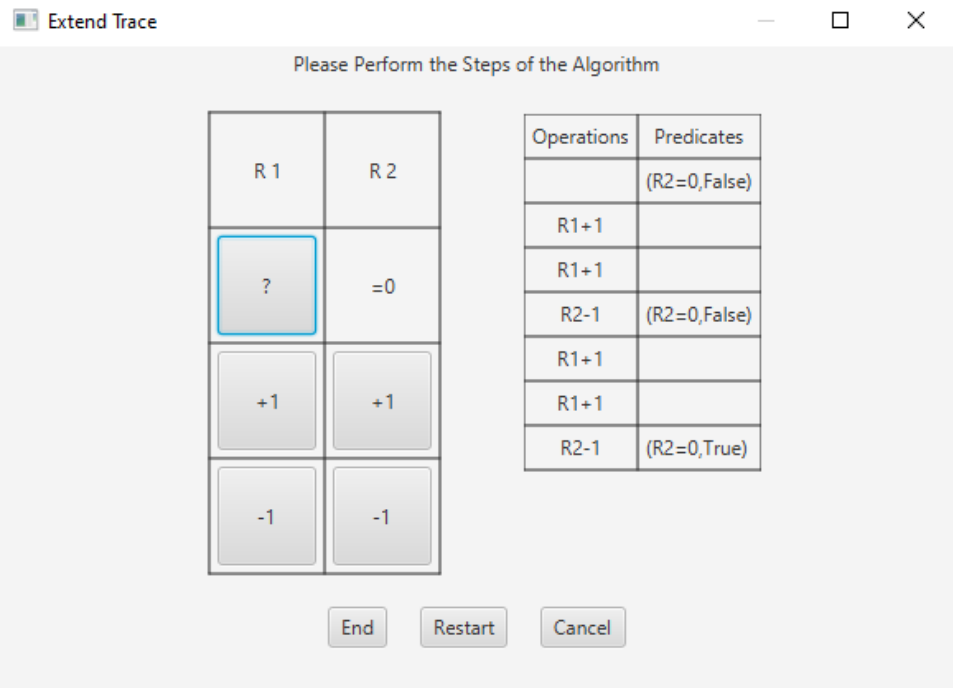

Figure 4.3.: Machine-Computer Game

- `TAB` navigates to the next cell to the right. If the current cell is the last column, the first cell in the next row is taken.
- `SHIFT+TAB` navigates to the previous cell to the left. If the current cell is in the first column, the last cell in the previous row is taken.
- `RETURN` or `DOWN ARROW` navigates to the cell below. If the current cell is in the last row, the first cell in the next column is taken.
- `SHIFT+RETURN` or `UP ARROW` navigates to the cell above. If the current cell is the first row, the last cell in the previous column is taken.

It is only possible to edit one cell at a time. However, it is possible to select multiple cells for deletion:

- Holding `CTRL` allows the user to select multiple cells by clicking the left mouse button on them. The cells do not have to be next to each other.
- To select a block of cells select one cell and then press `SHIFT` and left mouse click on another cell. This operation selects all cells in between the two cells.

Pressing the `DELETE` or `BACKSPACE` key erases the content of all selected cells. Pressing the right mouse button opens a menu with the options to insert or delete a row. Empty rows at the end of a trace are automatically deleted.

The user may also adjust session-independent settings. These include the text size of the machine program, the amount of displayed spaces per tab and if the GUI should automatically add quotations for SM machine states.

Only correctly formatted traces are displayed in a table. Traces may be malformed due to user error amidst editing trace files directly. A trace is malformed if it does not contain the correct number of columns in at least one row. The number of columns is determined by the MoC of the corresponding machine program file. Such traces are displayed in plain text to allow for error correction. After saving, the corrected version is displayed in a table. The GUI enforces the correct amount of columns, which implies that the described error cannot occur as long as the user only edits traces in the GUI.

4.3. Example: Multiplication Program

This section presents the complete development process of a machine program using trace-based programming with the GUI. The task is to implement an efficient algorithm for multiplication in a counter machine. It was first introduced by Chandoo to test trace-based programming using pen and paper. Algorithm 4.1 was developed in the process [Cha19]. Naturally, the presented steps are not the only viable solution.

The input of the algorithm is given in the registers one and two. The result of the multiplication is supposed to be written to register three. The MoC is a counter machine with four registers. Therefore, the algorithm has to compute the function

$$(x, y, \bullet, \bullet) \mapsto (\bullet, \bullet, x * y, \bullet)$$

where the dots represent any arbitrary values in $\mathbb{N}_0$.

Algorithm 4.1 Efficient Multiplication in the CM

```
1:  R3 ← 0
2:  R4 ← 0
3:  if R1 = 0 or R2 = 0 then End
4:  while True do
5:      R1 ← R1 − 1
6:      if R1 = 0 then
7:          R3 ← R3 + R2;  R2 ← 0
8:          End
9:      R3 ← R3 + R2;  R4 ← R2;  R2 ← 0
10:     R1 ← R1 − 1
11:     if R1 = 0 then
12:         R3 ← R3 + R4;  R4 ← 0
13:         End
14:     R3 ← R3 + R4;  R2 ← R4;  R2 ← 0
```

This is a good use case for trace-based programming because its general structure is simple. But, it is complicated to implement the program as there are numerous program states and long binary decision trees. That is the reason why trace-based programming is easier to use than ad-hoc programming in this case, because it provides a clear structure to the implementation process. Algorithm 4.1 displays the algorithm for multiplication.

The algorithm adds the value of $R2$ to $R3$ for every decrement of $R1$. It performs multiplication by repeated addition. It is efficient to shift the value of $R2$ between the registers two and four, because a machine program cannot store a non-constant amount of information. Therefore, one has to save the content of $R2$ in another register. It is in the nature of the MoC CM that the content of the addends is lost during addition. This is because the addition of two registers can only be performed by repeatedly incrementing the summand and decrementing the addend.

Suppose one already formed a mental representation of the algorithm. It is important to realize that the mental representation does not necessarily look like Algorithm 4.1. It could be a more abstract, simplified idea of the algorithm:

> *Firstly, empty register three and four. Secondly, decrement register one and add the value of register two to register three. Repeat step two of the process until register one is empty. The initial value of register two alternates between register two and four.*

Also, notice that this mental representation is not complete. It does not include edge cases like the condition checked in line 3 of Algorithm 4.1. Trace-based programming does not require the user to have a fully developed algorithm at hand as the edge cases become apparent during the process.

After opening the GUI, the development process begins with writing a fully undefined program. The MoC used is a CM with four registers. Machine Program 4.1 displays this initial program.

```
1 #MOC CM 4
2 #PROGRAM mult
3 Start:
4     Undef
5
6 Undef / NOP:
7     End
```

Machine Program 4.1: First Program Structure

The user enables the option to add traces in the GUI by saving the program. As the first step of the mental representation is the elimination of any values in register three and four it should also be the first feature to implement. It makes sense to start with a trace that only describes this behaviour. For example, one could add a trace with an initial machine state $(0, 0, 2, 2)$. Then, the user has to extend the trace by playing the

machine-computer game. Afterwards, one has to name the program states of the trace. There are two different program states required, one for emptying $R3$ and one for $R4$. It is recommended to use descriptive names, i.e. `EmptyR3` and `EmptyR4`. This yields Trace 4.1.

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		0	0	2	2	(R3=0,False)
EmptyR3	R3-1	0	0	1	2	(R3=0,False)
EmptyR3	R3-1	0	0	0	2	(R3=0,True),(R4=0,False)
EmptyR4	R4-1	0	0	0	1	(R4=0,False)
EmptyR4	R4-1	0	0	0	0	(R4=0,True)

Trace 4.1.: Empty R3 and R4

By extending the machine program with this trace it receives the two new states. At this point the first edge case becomes apparent. The values in $R1$ and $R2$ in the trace are both zero, which implies that the result of the multiplication is also zero and the computation has to end. Furthermore, it is only necessary to check if the value of $R1$ is zero. This is because the product of multiplication is zero if one of the factors is zero. Thus, the user has to start the machine-computer game again and add the predicate (R1=0,True) to the predicate sequence in the last row. Before extending the machine program again, one has to replace the directive `End` with `Undef` to insert the new predicate check. The result is displayed in Machine Program 4.3.

```
 1 #PROGRAM mult
 2 Start:
 3     R3=0
 4         EmptyR3
 5         Undef
 6
 7 EmptyR3 / R3-1:
 8     R3=0
 9         EmptyR3
10         R4=0
11             EmptyR4
12             Undef
13 EmptyR4 / R4-1:
14     R4=0
15         EmptyR4
16         R1=0
17             Undef
18             End
```

Machine Program 4.2: Partial Program after first Extension

Next, the user has to add a trace that includes the multiplication. For example, one could choose to add a trace with the initial machine state $(3, 1, 0, 0)$. This also includes the edge case of both of values in $R3$ and $R4$ being zero. While extending the trace, the user has to follow their mental representation of the algorithm. After decrementing $R1$ they have to shift the value in $R2$ to $R3$ and $R4$. The value in the first register is not

zero after a second decrement. Therefore, the value in $R4$ has to be shifted back to $R2$ and added to $R3$. Lastly, the value in $R2$ is zero after the third decrement. Thus, the user has to add the value in $R2$ to $R3$ one last time. Trace 4.2 is generated during this process. Again, the program states have to be named by the user. The program state `Dec1a` corresponds to line five in Algorithm 4.1. The states `Move2To4a/b/c` refer to line nine. Similarly, the state `Dec1b` belongs to line eleven and the states `Move4To2a/b/c` to line fourteen. The last two states `Add2To3a/b` resemble line seven of the algorithm. This mapping from program states to the algorithm is just presented for the reader. The user, however, only has a mental representation of the algorithm.

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		3	1	0	0	(R3=0,True),(R4=0,True),(R1=0,False), (R2=0,False)
Dec1a	R1-1	2	1	0	0	(R1=0,False)
Move2To4a	R2-1	2	0	0	0	
Move2To4b	R3+1	2	0	1	0	
Move2To4c	R4+1	2	0	1	1	(R2=0,True),(R1=0,False)
Dec1b	R1-1	1	0	1	1	(R1=0,False)
Move4To2a	R4-1	1	0	1	0	
Move4To2b	R3+1	1	0	2	0	
Move4To2c	R2+1	1	1	2	0	(R4=0,True),(R1=0,False)
Dec1a	R1-1	0	1	2	0	(R1=0,True)
Add2To3a	R2-1	0	0	2	0	
Add2To3b	R3+1	0	0	3	0	(R2=0,True)

Trace 4.2.: Three times One

Nearly all program states of the machine program exist after the user extends the program with Trace 4.2. This second partial program is displayed in Machine Program A.1 in the appendices. The only program states missing belong to the case where the initial multiplicand is in $R4$ when the value in $R1$ is decremented to zero (see line 12 of Algorithm 4.1). The user can generate a trace including this case by choosing an even value (greater than zero) for the multiplier in $R1$. To keep the traces short one should choose low values for the initial machine state, i.e. $(2, 1, 0, 0)$. Because the computation at the beginning of this trace is the same as the trace before, the user does not have to write the whole trace manually. They can automatically compute the first six rows by using the extend-trace function of the GUI. Afterwards, the user only has to add the last two rows manually.
The result is Trace 4.3. The two new program states `Add4To3a/b` refer to line thirteen of Algorithm 4.1. Extended with this trace, the machine program contains all program states needed. The resulting program is Machine Program A.2. However, some binary decision trees still have undefined subtrees. To eliminate them, the user has to retrace the program's execution and find initial machine states that lead to these subtrees. For

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		2	1	0	0	(R3=0,True),(R4=0,True),(R1=0,False), (R2=0,False)
Dec1a	R1-1	1	1	0	0	(R1=0,False)
Move2To4a	R2-1	1	0	0	0	
Move2To4b	R3+1	1	0	1	0	
Move2To4c	R4+1	1	0	1	1	(R2=0,True),(R1=0,False)
Dec1b	R1-1	0	0	1	1	(R1=0,True)
Add4To3a	R4-1	0	0	1	0	
Add4To3b	R3+1	0	0	2	0	(R4=0,True)

Trace 4.3.: Two times One

example, the BDT of the state `Start` has an undefined subtree at the end of the path (R3=0, True) and (R4=0,False). Because `Start` is the first state of a program, one can conclude that they have to add a trace where the initial value in $R3$ is zero, but the value in $R4$ is not, i.e. $(0, 0, 0, 1)$. The program's execution with this initial machine state should enter the state `EmptyR4` and then terminate. Thus, the user has to manually extend the trace. The trace is displayed in Trace B.1 in the appendices. After extending the machine program with the trace, the undefined subtree is eliminated.

Likewise, all other undefined subtrees have to be eliminated. The user can backtrack the program's execution to identify the necessary initial machine states. Then, they can use a combination of automatic and manual trace extensions to generate the traces. The following list presents an example of a combination of initial machine states, which generate traces necessary to complete the program for multiplication. All traces referenced can be found in the appendix.

- $(1, 0, 0, 0)$ and $(0, 1, 0, 0)$ yield Trace B.2 and Trace B.3 (see Appendix B). They eliminate the undefined subtrees in program state `Start`, representing the edge cases where one of the registers $R1$ or $R2$ is the only register with a non-zero value.
- $(0, 0, 1, 0)$, $(1, 0, 1, 0)$ and $(1, 1, 1, 0)$ can be used as initial machine states for traces that replace the undefined nodes in the BDT of state `EmptyR3` with defined subtrees (Trace B.4, B.5, and B.6).
- $(1, 0, 0, 1)$ and $(1, 1, 0, 1)$ can be used in a similarly manner for state `EmptyR4` (Trace B.7 and B.8).
- $(2, 2, 0, 0)$ yields Trace B.9, which eliminates undefined nodes in BDT of state `Move2To4c` and state `Add4To3`. This is because the value in $R2$ is greater than one and therefore it requires multiple loops to shift the value from $R2$ to $R4$.
- $(3, 2, 0, 0)$ eliminates undefined nodes in BDT of state `Move4To2c` and `Add2To3` in a similar manner (Trace B.10).

The program is complete after extending it with all traces mentioned. It is possible to achieve this with fewer traces by combining different edge cases. The complete program is presented as Machine Program A.3 in the appendices.

However, the complete program still has partial BDTs. The user can apply the procedure described in the proof of Theorem 3.5 to states `Move2To4c` and `Move4To2c`. This replaces the subtree beginning at node `R1=0` with the node `Dec1b` and `Dec1a`, respectively. The resulting program consists only of states with complete BDTs. Therefore, the undefined state is no longer needed and can be deleted. Machine Program A.4 displays the final program.

5. Implementation Details

The command line interface was developed using the Haskell 2010 programming language [Mar10]. The graphical user interface was programmed with Java 14.0.1 by Oracle and OpenJavaFX 14.0.1. It was necessary to use this version for the GUI because it introduces variable amounts of spaces per tab [jfx20]. This is essential for the proper formatting of machine programs.

5.1. Command Line Interface

The command line interface is divided into two modules. The main goal of this division is to separate pure and impure functions. The module `synth_cli` provides an interface to the user or the GUI. It mainly contains impure functions for user- or file-input and -output. The other module `Synthesizer` incorporates functions for consistency checks as well as machine program and trace extensions. Its functions are pure, as their results are returned to the functions of the `synth_cli` module.

The algorithms presented in this section display the abstract behaviour of implemented algorithms. Notably, the implemented functions use recursive instead of iterative structures, i.e. loops. This is because Haskell only allows for recursive structures. Nonetheless, the following algorithms use iterative structures for better readability.

The command line interface works with machine program and trace files. Therefore, it requires functions to parse machine programs and traces from a file into the data structures used by the underlying Haskell framework as well as converting them back. Haskell Programs 5.1, 5.2, and 5.3 display these data structures [Cha20a].

The data structure `Program` represents a machine program. It contains the binary decision tree of the start state. Additionally, it holds a mapping from program state names to tuples consisting of an operation name and a binary decision tree.

```
1 data Program = Program
2      LabeledTree
3      (Map ProgramState (OperationName,LabeledTree))
```

Haskell Program 5.1: Program Data Structure

All programs in a machine program file are encapsulated in the data structure `ParsedPrograms`. It comprises the name of the MoC used, the name of the first program in the file, and a mapping from program name to the `Program` and its argument. The argument is only used if the program supports function calls.

```
1 data ParsedPrograms = ParsedPrograms
2   ModelName
3   ProgramName
4   (Map ProgramName (Program,ProgramArgument))
```

Haskell Program 5.2: ParsedPrograms Data Structure

A trace is represented by a list of `TraceRow`s. Each `TraceRow` consists of the name of a program state, an optional operation, a string representing the machine state, the predicate sequence and the name of the next program state. Notably, the Haskell `TraceRow` includes the redundant information of the program state in the next trace row to facilitate computations.

```
1 data TraceRow = TraceRow
2   ProgramState              -- current program state
3   (Maybe OperationName)
4   MachineState
5   [(PredicateName,Bool)]
6   ProgramState              -- next program state
```

Haskell Program 5.3: TraceRow Data Structure

The most important functions of the Haskell framework are the parsing of a machine program file into a `ParsedPrograms` data structure and the execution of a `Program` to generate a list of `TraceRow`s. Additionally, it includes the definition of the MoCs presented in Chapter 2. Chandoo implemented these functions.

The framework only provides a method to parse the content of a machine program file into the Haskell representation. The other three functions of parsing `ParsedPrograms` into a machine program file, a list of `TraceRow`s into a trace file, and vice versa are implemented by the CLI.

The CLI provides three main functions to the user: extending machine programs, extending traces, and checking consistency. All three functions need similar data, thus it parses the command line arguments into a data structure called `ParsedInput`. This holds the programs of the machine program file, the name of the program to be used, a list of traces, and a flag to determine if consistency should be checked.
A `ParsedInput` is passed to intermediate functions, which extract the required information and pass it to one of the three main functions presented below. This intermediate functions also write the resulting output back to the machine program and trace files. Additionally, they display error messages to the user.

```
1 data ParsedInput = ParsedInput
2     ParsedPrograms
3     ProgramName
4     [[TraceRow]]
5     Bool              --check consistency
```

Haskell Program 5.4: ParsedInput Data Structure

5.1.1. Extend Machine Programs

The algorithm for extending a machine program Q from a set of traces T is shown in Algorithm 5.1. It takes a machine program Q and a list of traces T as an input and returns the extended program. Intuitively, the algorithm evaluates every trace row by row and checks if the execution of the program would enter undefined behaviour. If so, it extends the binary decision tree of the current program state. Additionally, the next program state has to be added to Q, if it does not exist yet. Therefore, the following functions are required:

- ExtendBDT(n, f) alters the binary decision tree of program state p by adding the information given in predicate sequence f.
- AddPState(Q, n, o) adds a new program state labelled n with operation o. Its binary decision tree consists of only a root node containing the directive to enter the undefined state.

The traces $t \in T$ have the format displayed in Haskell Program 5.3. Therefore, the variable n_i corresponds to the current program state in trace row i. o_i corresponds to the operation, s_i to the machine state, f_i to the predicate sequence, and m_i to the next program state. The term $Q^1(n_i, s_i)$ refers to the execution of machine program Q starting in program state n_i with the initial machine state s_i for exactly one step. This yields one trace row. The term $t[i]$ accesses the trace row at index i in trace t.
Algorithm 5.1 does not display the error and consistency checks performed by the implemented function because they contain many edge cases. A non-exhaustive list is presented below:

- $\forall\, 0 \leq i \leq |t| : n_i \neq$ `"Undef"` $\wedge\, m_i \neq$ `"Undef"` : Traces $t \in T$ are not allowed to have undefined states. This was a design decision to ensure that traces never terminate earlier than the program. It enforces a clear separation between the completely defined behaviour in traces and the possibly undefined behaviour of the machine program.
- $\forall\, 0 \leq i \leq |t| : n_i = n_i' \wedge o_i = o_i' \wedge s_i = s_i'$: The variables o_i', p_i', s_i' refer to the variables computed in line six in the i-th iteration of the inner loop. If one of the equalities does not hold, then the trace involves different behaviour than the machine program. Therefore, the machine program cannot be extended. Operations

Algorithm 5.1 Extend Machine Programs

```
1: function ExtendMP(Q, T)
2:     for t ∈ T do
3:         for i = 0 ... |t| do
4:             (n_i, o_i, s_i, f_i, m_i) = t[i]
5:             (n', o', s', f', m') ← Q^1(n_i, s_i)
6:             if m' = "Undef" then
7:                 ExtendBDT(n_i, f_i)
8:                 if m_i not exists in Q and i ≠ |t| then
9:                     AddPState(Q, m_i, o_{i+1})
10:    return Q
```

o_0 and o'_0 are mathematically undefined (see Section 3.2), which is represented by the value `Nothing` in Haskell [Mar10]. It is necessary to check that both variables embody this value.

- $\forall\, 0 \leq i \leq |t| : f_i = f'_i \vee (m'_i = \texttt{"Undef"} \wedge f_i =_k f'_i)$, with $k = |f'_i|$: The variables f'_i and n'_i again refer to the i-th iteration of the inner loop. The symbol $=_k$ refers to the equality of the first k items. If the program state in the next row is not the undefined state, then the predicate sequences of the given and the evaluated trace have to be equal. Otherwise, the trace rows only have to be equal up to the length of the evaluated predicate sequence.

- $|T| > 0$: The program cannot be extended if there are no traces.

5.1.2. Extend Traces

The second important function of the command line interface is extending traces. The implemented algorithm is displayed in Algorithm 5.2. It takes a machine program Q and a single trace t as an input. The algorithm returns the extended trace or an error if the trace cannot be extended with the given machine program. It checks for every trace row in the input trace if it is equal to the trace row generated by the program. If the generated trace terminates in the same row as the given trace, then only the last trace row needs to be updated. If the generated trace is longer than the input trace, then it simply returns the evaluated trace.

The term $Q^*(p_0, s_0)$ refers to the execution of Q with initial machine state s_0 and program state n_0 until it terminates. Because this computation is possibly infinitely long, it is not guaranteed that Algorithm 5.2 terminates. This behaviour is unavoidable because the machine program may contain infinite loops and the halting problem is undecidable [Tur37]. However, the user may interrupt the algorithm when using the CLI by pressing `CTRL+C`. The GUI uses a timeout of five seconds to interrupt long computations.

The decision to allow the predicate sequence of the last trace row to be empty, even if the machine program checks predicates at this point of evaluation, was by design. This allows users to extend traces when only the initial machine state is known. This is essential because it enables the user to generate traces of the machine program's execution by only specifying the initial machine state. One does not have to think about the predicates checked or the behaviour of the program in general.

Algorithm 5.2 Extend a Trace

```
1: function ExtendTrace(Q, t)
2:     (n_0, o_0, s_0, f_0, m_0) ← t[0]
3:     t' ← P*(n_0, s_0)
4:     if |t| > |t'| then
5:         Error
6:     for i = 0 ... |t| do
7:         (n_i, o_i, s_i, f_i, m_i) ← t[i]
8:         (n'_i, o'_i, s'_i, f'_i, m'_i) ← t'[i]
9:         if i < |t| and t[i] ≠ t'[i] then
10:            Error
11:        if i = |t| then
12:            if n_i = n'_i and o_i = o'_i and s_i = s'_i and (f_i = f'_i or |f_i| = 0) then
13:                t[i] ← t'[i]
14:                if m'_i = "End" then
15:                    return t
16:            else
17:                Error
18:        if i = |t| + 1 then
19:            return t'
```

5.1.3. Consistency

The last function allows the user to test consistency between a machine program Q and a set of traces T. It implements Definition 3.9 as a consistency check. Algorithm 5.3 presents an abbreviated version of the algorithm used. The complete algorithm includes further checks to ensure that the trace is formatted properly. It returns a boolean value.

The term $\neq_{|f'_i|}$ refers to inequality up to the length of f'_i. Intuitively, the algorithm checks for every row of every trace in T if it fits to the trace row generated by executing Q. A detailed explanation of the conditions checked is given below:

– $|t| < |t'| - 1$: If the input trace is shorter than the evaluated trace without the last row, then it ends before the machine program terminates.

– $n_i \neq n'_i \vee o_i \neq o'_i \vee s_i \neq s'_i$: The name of the program state, the operation, and the machine state of every row have to be equal (see Definition 3.9).

– $f_i \neq_{|f'_i|} f'_i \vee \left(m_i \neq \texttt{"Undef"} \wedge f_i \neq f'_i\right)$: The predicate sequences of the traces have to be equal or, if the next program state is undefined, equal up to the length of the generated predicate sequence (See Definition 3.9).

– $m'_i = \texttt{"Undef"}$: If all of the conditions above are false and the next program state is undefined, then the trace is consistent.

– $\left(m_i = \texttt{"End"} \wedge m'_i \neq \texttt{"End"}\right) \vee \left(m_i \neq \texttt{"End"} \wedge m'_i = \texttt{"End"}\right)$: If the trace ends at a different point than the program's execution, then the trace cannot be consistent with the machine program.

Algorithm 5.3 Consistency

```
1:  function CONSISTENT(Q, T)
2:      for t ∈ T do
3:          (n_0, o_0, s_0, f_0, m_0) ← t[0]
4:          t' ← Q*(n_0, s_0)
5:          if |t| < |t'| − 1 then
6:              return False
7:          for i = 0 ... |t| do
8:              (n_i, o_i, s_i, f_i, m_i) ← t[i]
9:              (n'_i, o'_i, s'_i, f'_i, m'_i) ← t'[i]
10:             if n_i = "Undef" or m_i = "Undef" then return False
11:             if n_i ≠ p'_i or o_i ≠ o'_i or s_i ≠ s'_i then return False
12:             if f_i ≠_{|f'_i|} f'_i or (m_i ≠ "Undef" and f_i ≠ f'_i) then return False
13:             if m'_i = "Undef" then break
14:             if (m_i = "End" and m'_i ≠ "End") or (m_i ≠ "End" and m'_i = "End")
    then
15:                 return False
16:     return True
```

5.2. Interface between CLI and GUI

The main purpose of the GUI is to present the functions of the CLI more conveniently. Therefore, it needs to access the functions of the CLI. The only alternative would be to

implement these functions directly in the GUI. Some of the functions presented would not require great effort to be implemented in the GUI. However, this would lead to duplicate code, which complicates changes and thereby is considered bad programming practice [BF99].

In addition to the functions discussed in Section 4.1 the CLI provides an interface with five more functions. The GUI accesses the functions of consistency checking, extending a program, and extending a trace in the same way as described in Section 4.1. Thus, only presents the other five functions. They are accessed with the following CLI command:

```
synth_cli.exe id arguments...
```

The argument `id` identifies the function to be used. It always begins with two dashes. The second part `arguments...` refers to a list of arguments provided as input for the function where the number of arguments varies between the functions. All functions write their return value to the standard output file descriptor. If an error occurs it is written to the standard error stream and no value is returned to the standard output.

The first function computes the number of registers used by a model of computation. This function is necessary to determine the proper amount of columns in a trace. It takes the name of an MoC as an argument. The return value is an integer representing the number of registers used by the given MoC. It can be accessed with the command:

```
synth_cli.exe --nreg model
```

The next function determines the model of computation of a given model name. The GUI uses it to provide different user interfaces given different models of computation. I.e., the option to extend a trace manually is only available, if the current MoC is a counter machine. Input of the function is a model name and the return value a string identifier of the model. The three possible identifiers are `CM` for a counter machine, `SM` for a stack machine, and `GENERIC` for all other generic models. The syntax of the function call is:

```
synth_cli.exe --moc model
```

The next two functions are essential for extending traces manually. The GUI has to check predicates and determine the effects of operations for the current machine state to compute a trace row. Therefore, the next function computes the new machine state when applying an operation of a given MoC to a machine state. It returns the new machine state in a csv-representation, which is a string of values separated by semicolons. The function is accessed with the command:

```
synth_cli.exe --operation model state op
```

The argument `model` is a model name (e.g. `"SM 2 01"`), which is not to be confused with the identifiers discussed in the last function (e.g. `SM`). The `state` is a Haskell representation of the machine state. The last argument `op` is a string containing the

operation. For example, a function call using a counter machine with two registers could look like:

```
synth_cli.exe --operation "CM 2" "[0,1]" "R1+1"
"1;1"
```

The next function checks a predicate for any given machine state and MoC. It returns either a zero, if the predicate is false, or a one, if the predicate is true. It uses a syntax similar to the last function. The argument `pred` is a string containing the predicate to check:

```
synth_cli.exe --operation model state pred
```

The last function is necessary for the machine-computer game of a stack machine in the GUI. In contrast to a counter machine, the predicates and operations of a stack machine depend on the input alphabet. The CLI offers the following function to retrieve the input alphabet of a stack machine:

```
synth_cli.exe --ia model
```

It is also possible to use the function with a Turing machine; however, this is currently not necessary.

5.3. Graphical User Interface

The GUI is a stateless interface in a sense that it does not generate a session state [DHJP08]. Its output is the graphical data displayed to the user, which only depends on the files used and the input given by the user. However, the GUI does save information about the machine program to reduce the amount of file input. For example, it saves the MoC and the number of registers used.

Figure 5.1 shows the main data flow between GUI, CLI, and files. The GUI loads the machine program and trace files and displays them to the user. Then, the user may change the content of the machine program or the traces by editing them. These changes are saved to the files. As an alternative, the user may also call the CLI, which directly changes the file content. Afterwards, the GUI reloads the content and displays it to the user.

The main stage of the GUI is divided into three parts: An area for a machine program, traces, and the console. This division is reflected by the modularization of the control structures. Hence, the main stage has four controllers to manage the elements of the user interface. This includes one controller for each of the three areas and a primary controller to connect all functionality.

The text of the console is saved in a dedicated log file. This is useful because the error stream of a command line interface process can be appended to this file. The console itself is a read-only text area, which is synchronized with the log file by the controller of the console. Also, the controller can append messages generated by the GUI to the log

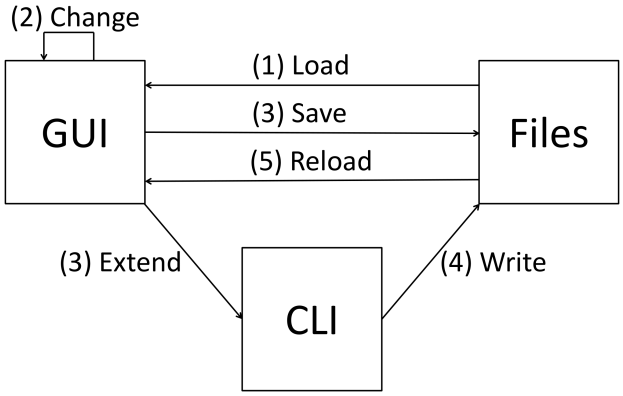

Figure 5.1.: Composition of Interfaces and Files

file, e.g. if the user deletes a trace. Whenever new content is displayed in the console, the controller automatically scrolls down to the new content.

The session-independent preferences of the user are saved in a preference file. This is a text file containing key-value pairs. Every pair is located in a new line:

```
key: value
```

The keys `text-size` and `tab-size` correspond to an integer value. But, the key `add-quotation` refers to a lower-case boolean value, e.g. `true`.

The machine program area consists mainly of an editable text area displaying the content of the machine program file. Furthermore, there are buttons to load, extend, and save the machine program. The machine program file is synchronized if the user clicks one of the dedicated buttons, e.g. the save button. A drop-down menu allows the user to select one of the programs in the machine program file. The controller of the machine program handles the synchronization of the machine program file with the text area. Additionally, in handles the button callbacks as well as the file hierarchy. This functionality is located in the controller of the machine program file because the machine program file is the file on the highest level of the hierarchy. Therefore, it can be considered the anchor of the hierarchy. The position of all other parts of the hierarchy are computed with respect to the machine program file.

Traces are presented as tabs in a tab pane. The trace controller handles the tab pane as well as the callbacks of buttons, which alter the content of traces. These are the buttons for adding, saving, deleting, and extending traces. Similar to the machine program area, some buttons are located in the menu bar at the top to improve the user experience. Each tab in the tab pane is associated with one trace file. Only tabs belonging to the currently selected program in the machine program area are displayed. Every time the user selects a different program all tabs have to be reloaded.

The items of a trace are arranged in a table, assuming that the trace is properly formatted. If a trace contains an invalid header line or the wrong amount of columns it is displayed as plain text. The amount of columns is determined by the MoC of the corresponding machine program file. Most buttons are disabled if one of the traces is

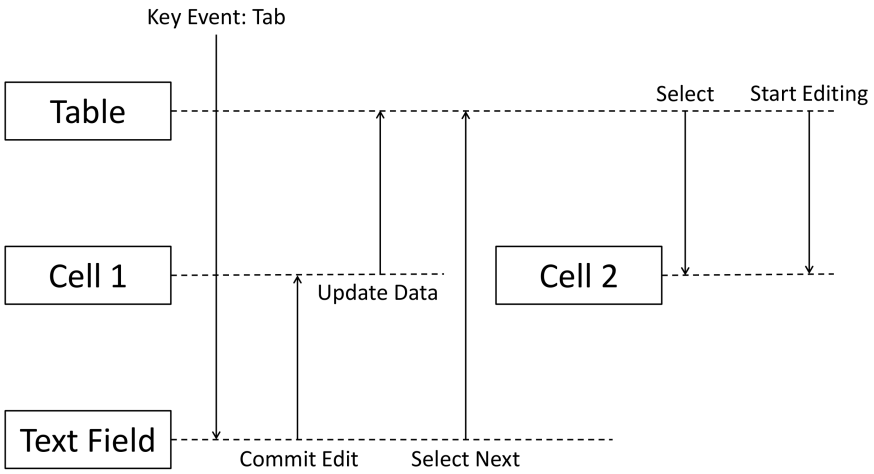

Figure 5.2.: Event Flow after Tabulator-Key Pressed

malformatted because their functions depend on the trace data. The cells in a table contain either the default text display of a JavaFX-TableCell or a custom text field if the user is editing the content of the cell. The trace data is updated with the content of the text field every time the user commits a change in one of the cells. It requires a complex structure of event messages to detect if a user has finished editing a cell. This is because every cell handles user induced changes independently.

Figure 5.2 presents an example for the event flow if a user presses the tabulator key while editing a cell. As described in Section 4.2, this directly selects the next cell to the right. The event flow begins with a key event caused by the user pressing the tab key while editing the content of the text field. This event can only be noticed by the text field as it is the currently focused user interface element. The text field signals the cell that the user stopped editing and that the changes made are ready to be committed. The cell signals the table to update its data with the new data. Furthermore, the text field sends a custom event to the table indicating it should select the next cell and start editing its content.

6. Conclusion and Future Work

In this thesis, I presented two user interfaces for trace-based programming. They introduce novice programmers to trace-based programming, which separates the algorithm design from the implementation process [Cha20a]. This can be accomplished by playing the algorithm in a machine-computer game and automatically synthesizing a machine program from the generated traces.

The main use is intended to be in introductory programming courses. This is because the major benefit of the method is that a user does not need any experience in a specific programming language. It could be utilized in highschools or even basic programming classes at universities. However, the method and the user interfaces have to be tested first. As of now, Chandoo discovered promising results of the method itself in a test with two tenth-graders [Cha20a]. That being said, the usability of the interfaces has not been tested yet.

An option to use subprogram or function calls as operations would improve the machine-computer game. This would allow the user to split a given task into smaller subtasks and implement them individually by using only trace-based programming.

Additionally, future work could include adding a simulation of the machine-computer game for Turing machines to the GUI. This could enable the use of the method in theoretical computer science classes. On the contrary, the syntax used for Turing machines in computer science classes likely differs from the syntax of machine programs. However, if students are given the task of creating a Turing machine program, trace-based programming could be used to facilitate the development process. This is because the difficult part about writing Turing machine programs presumably is implementing the program, not creating a mental representation of the algorithm. Large-scale tests are, however, required to verify the described utility.

List of Figures

List of Programs

List of Traces

List of Algorithms

Bibliography

[BF99] Kent Beck and Martin Fowler. *Refactoring: Improving the Design of Existing Code*, pages 75–76. Addison Wesley, January 1999.

[Cha19] Maurice Chandoo. *Games and Interpreter for Counter and Stack Machines*. `https://upsl.uber.space/aws19/info.txt`, 2019. Accessed: 2020-07-15.

[Cha20a] Maurice Chandoo. Separating Algorithmic Thinking and Programming. *Institutionelles Repositorium der Leibniz Universität Hannover*, 2020.

[Cha20b] Maurice Chandoo. A Systematic Approach to Programming. *CoRR*, abs/1808.08989v3, 2020.

[DHJP08] Laurent Doyen, Thomas A. Henzinger, Barbara Jobstmann, and Tatjana Petrov. Interface Theories with Component Reuse. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT 08, pages 79–88, New York, NY, USA, 2008. Association for Computing Machinery.

[jfx20] JavaFX 14 Documentation. `https://openjfx.io/javadoc/14/`, 2020. Accessed: 2020-06-28.

[Lam61] Joachim Lambek. How to Program an Infinite Abacus. *Canadian Mathematical Bulletin*, pages 295 – 302, 1961.

[Mar10] Simon Marlow. *Haskell 2010 Language Report*, July 2010.

[Min67] Marvin Lee Minsky. *Computation, Finite And Infinite Machines*. Prentice-Hall series in automatic computation. Prentice-Hall, 1967.

[Sor13] Juha Sorva. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education*, June 2013.

[Tur37] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, S2-42(1):230–265, January 1937.

[Upa10] Muhammad Afzal Upal. An alternative account of the minimal counterintuitiveness effect. *Cognitive Systems Research*, 11(2):194–203, 2010.

A. Machine Programs

Machine Program A.1: Partial Program missing two States

```
#MOC CM 4


#PROGRAM mult
Start:
    R3=0
        EmptyR3
        R4=0
            Undef
            R1=0
                R2=0
                    Dec1a
                    Undef
                Undef


EmptyR3 / R3-1:
    R3=0
        EmptyR3
        R4=0
            EmptyR4
            Undef

EmptyR4 / R4-1:
    R4=0
        EmptyR4
        R1=0
            Undef
            End

Dec1a / R1-1:
    R1=0
        Move2To4a
        Add2To3a

Move2To4a / R2-1:
    Move2To4b

Move2To4b / R3+1:
    Move2To4c

Move2To4c / R4+1:
    R2=0
        Undef
        R1=0
            Dec1b
            Undef

Move4To2a / R4-1:
    Move4To2b

Move4To2b / R3+1:
    Move4To2c

Move4To2c / R2+1:
    R4=0
        Undef
        R1=0
            Dec1a
            Undef

Dec1b / R1-1:
    R1=0
        Move4To2a
        Undef

Add2To3a / R2-1:
    Add2To3b

Add2To3b / R3+1:
    R2=0
        Undef
        End

Undef / NOP:
    End
```

Machine Program A.2: Partial Program with all States

```
#MOC CM 4


#PROGRAM mult
Start:
    R3=0
        EmptyR3
        R4=0
            Undef
            R1=0
                R2=0
                    Dec1a
                    Undef
                Undef

EmptyR3 / R3-1:
    R3=0
        EmptyR3
        R4=0
            EmptyR4
            Undef

EmptyR4 / R4-1:
    R4=0
        EmptyR4
        R1=0
            Undef
            End

Dec1a / R1-1:
    R1=0
        Move2To4a
        Add2To3a

Move2To4a / R2-1:
    Move2To4b

Move2To4b / R3+1:
    Move2To4c



Move2To4c / R4+1:
    R2=0
        Undef
        R1=0
            Dec1b
            Undef

Dec1b / R1-1:
    R1=0
        Move4To2a
        Add4To3a

Move4To2a / R4-1:
    Move4To2b

Move4To2b / R3+1:
    Move4To2c

Move4To2c / R2+1:
    R4=0
        Undef
        R1=0
            Dec1a
            Undef

Add2To3a / R2-1:
    Add2To3b

Add2To3b / R3+1:
    R2=0
        Undef
        End

Add4To3a / R4-1:
    Add4To3b

Add4To3b / R3+1:
    R4=0
        Undef
        End

Undef / NOP:
    End
```

Machine Program A.3: Mutliplication with partial BDTs

```
#MOC CM 4


#PROGRAM mult
Start:
    R3=0
        EmptyR3
        R4=0
            EmptyR4
            R1=0
                R2=0
                    Dec1a
                    End
                End


EmptyR3 / R3-1:
    R3=0
        EmptyR3
        R4=0
            EmptyR4
            R1=0
                R2=0
                    Dec1a
                    End
                End


EmptyR4 / R4-1:
    R4=0
        EmptyR4
        R1=0
            R2=0
                Dec1a
                End
            End


Dec1a / R1-1:
    R1=0
        Move2To4a
        Add2To3a

Move2To4a / R2-1:
    Move2To4b

Move2To4b / R3+1:
    Move2To4c

Move2To4c / R4+1:
    R2=0
        Move2To4a
        R1=0
            Dec1b
            Undef

Dec1b / R1-1:
    R1=0
        Move4To2a
        Add4To3a

Move4To2a / R4-1:
    Move4To2b

Move4To2b / R3+1:
    Move4To2c

Move4To2c / R2+1:
    R4=0
        Move4To2a
        R1=0
            Dec1a
            Undef

Add4To3a / R4-1:
    Add4To3b

Add4To3b / R3+1:
    R4=0
        Add4To3a
        End

Add2To3a / R2-1:
    Add2To3b

Add2To3b / R3+1:
    R2=0
        Add2To3a
        End

Undef / NOP:
    End
```

Machine Program A.4: Mutliplication

```
#MOC CM 4


#PROGRAM mult
Start:
    R3=0
        EmptyR3
        R4=0
            EmptyR4
            R1=0
                R2=0
                    Dec1a
                    End
                End


EmptyR3 / R3-1:
    R3=0
        EmptyR3
        R4=0
            EmptyR4
            R1=0
                R2=0
                    Dec1a
                    End
                End


EmptyR4 / R4-1:
    R4=0
        EmptyR4
        R1=0
            R2=0
                Dec1a
                End
            End

Dec1a / R1-1:
    R1=0
        Move2To4a
        Add2To3a

Move2To4a / R2-1:
    Move2To4b

Move2To4b / R3+1:
    Move2To4c

Move2To4c / R4+1:
    R2=0
        Move2To4a
        Dec1b

Dec1b / R1-1:
    R1=0
        Move4To2a
        Add4To3a

Move4To2a / R4-1:
    Move4To2b

Move4To2b / R3+1:
    Move4To2c

Move4To2c / R2+1:
    R4=0
        Move4To2a
        Dec1a

Add4To3a / R4-1:
    Add4To3b

Add4To3b / R3+1:
    R4=0
        Add4To3a
        End

Add2To3a / R2-1:
    Add2To3b

Add2To3b / R3+1:
    R2=0
        Add2To3a
        End
```

B. Traces

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		0	0	0	1	(R3=0,True),(R4=0,False)
EmptyR4	R4-1	0	0	0	0	(R4=0,True),(R1=0,True)

Trace B.1.: Multiplication Example: $(0, 0, 0, 1)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		1	0	0	0	(R3=0,True),(R4=0,True),(R1=0,False), (R2=0,True)

Trace B.2.: Multiplication Example: $(1, 0, 0, 0)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		0	1	0	0	(R3=0,True),(R4=0,True),(R1=0,True)

Trace B.3.: Multiplication Example: $(0, 1, 0, 0)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		0	0	1	0	(R3=0,False)
EmptyR3	R3-1	0	0	0	0	(R3=0,True),(R4=0,True),(R1=0,True)

Trace B.4.: Multiplication Example: $(0, 0, 1, 0)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		1	0	1	0	(R3=0,False)
EmptyR3	R3-1	1	0	0	0	(R3=0,True),(R4=0,True),(R1=0,False), (R2=0,True)

Trace B.5.: Multiplication Example: $(1, 0, 1, 0)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		1	1	1	0	(R3=0,False)
EmptyR3	R3-1	1	1	0	0	(R3=0,True),(R4=0,True),(R1=0,False), (R2=0,False)
Dec1a	R1-1	0	1	0	0	(R1=0,True)
Add2To3a	R2-1	0	0	0	0	
Add2To3b	R3+1	0	0	1	0	(R2=0,True)

Trace B.6.: Multiplication Example: $(1, 1, 1, 0)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		1	0	0	1	(R3=0,True),(R4=0,False)
EmptyR4	R4-1	1	0	0	0	(R4=0,True),(R1=0,False),(R2=0,True)

Trace B.7.: Multiplication Example: $(1, 0, 0, 1)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		1	1	0	1	(R3=0,True),(R4=0,False)
EmptyR4	R4-1	1	1	0	0	(R4=0,True),(R1=0,False),(R2=0,False)
Dec1a	R1-1	0	1	0	0	(R1=0,True)
Add2To3a	R2-1	0	0	0	0	
Add2To3b	R3+1	0	0	1	0	(R2=0,True)

Trace B.8.: Multiplication Example: $(1, 1, 0, 1)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		2	2	0	0	(R3=0,True),(R4=0,True),(R1=0,False), (R2=0,False)
Dec1a	R1-1	1	2	0	0	(R1=0,False)
Move2To4a	R2-1	1	1	0	0	
Move2To4b	R3+1	1	1	1	0	
Move2To4c	R4+1	1	1	1	1	(R2=0,False)
Move2To4a	R2-1	1	0	1	1	
Move2To4b	R3+1	1	0	2	1	
Move2To4c	R4+1	1	0	2	2	(R2=0,True),(R1=0,False)
Dec1b	R1-1	0	0	2	2	(R1=0,True)
Add4To3a	R4-1	0	0	2	1	
Add4To3b	R3+1	0	0	3	1	(R4=0,False)
Add4To3a	R4-1	0	0	3	0	
Add4To3b	R3+1	0	0	4	0	(R4=0,True)

Trace B.9.: Multiplication Example: $(2, 2, 0, 0)$

State	Op.	R1	R2	R3	R4	Predicate Sequence
Start		3	2	0	0	(R3=0,True),(R4=0,True),(R1=0,False), (R2=0,False)
Dec1a	R1-1	2	2	0	0	(R1=0,False)
Move2To4a	R2-1	2	1	0	0	
Move2To4b	R3+1	2	1	1	0	
Move2To4c	R4+1	2	1	1	1	(R2=0,False)
Move2To4a	R2-1	2	0	1	1	
Move2To4b	R3+1	2	0	2	1	
Move2To4c	R4+1	2	0	2	2	(R2=0,True),(R1=0,False)
Dec1b	R1-1	1	0	2	2	(R1=0,False)
Move4To2a	R4-1	1	0	2	1	
Move4To2b	R3+1	1	0	3	1	
Move4To2c	R2+1	1	1	3	1	(R4=0,False)
Move4To2a	R4-1	1	1	3	0	
Move4To2b	R3+1	1	1	4	0	
Move4To2c	R2+1	1	2	4	0	(R4=0,True),(R1=0,False)
Dec1a	R1-1	0	2	4	0	(R1=0,True)
Add2To3a	R2-1	0	1	4	0	
Add2To3b	R3+1	0	1	5	0	(R2=0,False)
Add2To3a	R2-1	0	0	5	0	
Add2To3b	R3+1	0	0	6	0	(R2=0,True)

Trace B.10.: Multiplication Example: $(3, 2, 0, 0)$