PROGRESS IN NATURAL SCIENCE Vol. 17, No. 7, July 2007

SHORT COMMUNICATIONS

A language for easy and efficient modeling of Turing machines

Pinaki Chakraborty*

(School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi 110067, India)

Accepted on January 31, 2007

Abstract A Turing Machine Description Language (TMDL) is developed for easy and efficient modeling of Turing machines. TMDL supports formal symbolic representation of Turing machines. The grammar for the language is also provided. Then a fast single-pass compiler is developed for TMDL. The scope of code optimization in the compiler is examined. An interpreter is used to simulate the exact behavior of the compiled Turing machines. A dynamically allocated and resizable array is used to simulate the infinite tape of a Turing machine. The procedure for simulating composite Turing machines is also explained. In this paper, two sample Turing machines have been designed in TMDL and their simulations are discussed. The TMDL can be extended to model the different variations of the standard Turing machine.

Keywords: Turing machine, compiler, interpreter, simulation.

Turing machines are the most generally constructed models of computation[1]. They are widely used in the study of the concept of formal languages and the theory of computation. Despite of their importance in theoretical computer science, not a great deal of work has been done on the simulation of Turing machines. The Tutor Turing machine simulator[2] is a competent tool where the Turing machines are represented by flow diagrams. But it is intricate and not suitable for novice users. Hence, there is a need of a simple technique to study the working of Turing machines.

In this paper a simple language for modeling Turing machines is developed. The language is called the Turing Machine Description Language (TMDL). In TMDL, a Turing machine is represented in a way similar to that in any standard text. A compiler is then developed to translate a Turing machine written in TMDL to an Intermediate Language (IL). Finally, an interpreter is used to simulate the functioning of the compiled Turing machine.

1 The language

The TMDL is a simple language that can be used to design all possible Turing machines. It supports formal symbolic representation of Turing machines. No flow graph or Hernes table is required. The TMDL gains its advantage by modeling Turing machines far more efficiently than other languages. A TMDL program contains only the necessary information about a Turing machine and that also in a symbolic form. The TMDL is neither a procedural language nor an object oriented language as it does not implement any algorithm or instantiate any object. It is similar to hardware description languages except that the hardware of a Turing machine is virtual. Using this language requires no programming skill or any in depth study of the language specifications. So, it is advantageous for students and other naive users. A TMDL program can be divided into two major parts—a Turing machine definition and a transition function definition. The TMDL grammar is as follows.

```
program→tm_definition tf_definition;
tm_definition→tm = ({state_list}, {symbol_
              list}, {tape_symbol_list},
              transition_function, state,
              tape_symbol, {accepting_
              state_list});
tf_definition→transitions | ε
state_list→state, state_list | state
symbol_list→symbol, symbol_list | symbol
tape_symbol_list→tape_symbol, tape_symbol_
```

* E-mail: pinaki_chakraborty_163@yahoo.com

```
                    list | tape_symbol
accepting_state_list→state_list | ε
transitions → transition_function ( state,
              symbol ) = ( state, tape_sym-
              bol, direction), transitions |
              transition_function ( state,
              symbol ) = ( state, tape_sym-
              bol, direction)
```

In this grammar, **tm** stands for the keyword *TM*. A **state** is a sequence of lowercase letters, digits and underscore (_) characters starting with a letter and having a maximum of 8 characters. A **symbol** is any lowercase letter or any digit. A **tape _ symbol** is either a **symbol** or an underscore (_) character. A **transition _ function** is an uppercase letter except *L* and *R*. A **direction** is either *L* or *R*.

2 The compiler

2.1 The design

The TMDL compiler (TMDLC) is a fast single-pass compiler[3,4] written in C++. It takes as input a TMDL program which is a description of a Turing machine and produces as output a functionally equivalent program in the IL. Therefore, the TMDLC can be represented as $C_{C++}^{\text{TMDL IL}}$. The TMDLC consists of four phases, viz., lexical analyzer, syntax analyzer, semantic analyzer and the code generator (Fig. 1). Apart from these four phases, there is a bookkeeping module and an error handler module. The syntax analyzer plays the leading role in the process of compilation and the other three passes run as co-routines in the hegemony of the syntax analyzer. The syntax analyzer calls the lexical analyzer whenever the former needs a token. The lexical analyzer returns the next token in the input program on being called. On successful syntax analysis of a part of a program, the semantic analyzer and then the code generator are invoked for semantic analysis and code generation respectively. The syntax analyzer used in the TMDLC is a top-down predictive parser. The parser can recognize the next production rule to be used in the derivation by observing only the next token in the input stream. To design such a parser, the TMDL grammar is required to be converted to its LL(1) equivalent. The bookkeeping module maintains a symbol table to store the names of the states of the Turing machine. The names are inserted in the symbol table by the lexical analyzer. The semantic analyzer and the code generator use the information stored in the symbol table. The error handler used in the TMDLC is a trivial one. On detecting an error, it generates an error message and stops the process of compilation. The error messages are descriptive and help in debugging the program. The error handler is called by the lexical analyzer, the syntax analyzer and the semantic analyzer on the occurrence of lexical error, syntax error and semantic error respectively.

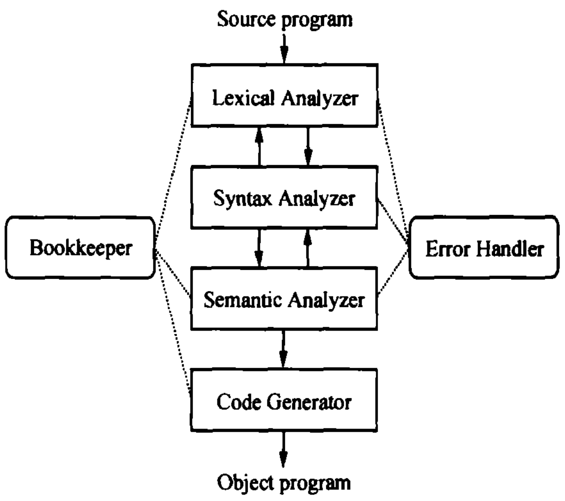

Fig. 1. Block diagram of the TMDLC.

2.2 Scope of code optimization

A code optimization phase can be included just before the code generator in the TMDLC. There are three key sources of optimization—useless tape symbols, useless states and useless transitions. A non-blank tape symbol, that is not an element of the input alphabet and not written back on the tape by any transition, will never appear on the tape and hence it is useless. A state that is not reachable from the initial state for any input string is useless as it is not possible for the Turing machine to be in that state. A transition that is defined in terms of either a useless tape symbol or a useless state is also useless. For every Turing machine $\boldsymbol{T} = (\boldsymbol{Q}, \boldsymbol{\Sigma}, \boldsymbol{\Gamma}, \delta, q_0, \square, \boldsymbol{F})$ there exist a Turing machine $\boldsymbol{T}'$ such that $\boldsymbol{T}'$ does not contain useless tape symbols, useless states and useless transitions and $\boldsymbol{L}(\boldsymbol{T}') = \boldsymbol{L}(\boldsymbol{T})$.

Detection of useless tape symbols:

A tape symbol $ts_1 \in \boldsymbol{\Gamma}$ is useless if and only if $ts_1 \neq \square$, $ts_1 \in \boldsymbol{\Sigma}$ and there exists no transition of the form $\delta(q_i, ts_k) = (q_j, ts_1, dir)$ where $ts_k \in \boldsymbol{\Gamma}$ and $dir \in \{L, R\}$.

Detection of useless states:

A null graph with the vertices labeled by the states of the Turing machine is drawn. A directed edge (q_i, q_j) is added to the graph for the transition $\delta(q_i, ts_1) = (q_j, ts_2, dir)$, where $ts_1, ts_2 \in \boldsymbol{\Gamma}$ and $dir \in \{L, R\}$. A state q_k is useless if and only if there exists no directed path from the vertex labeled q_0 to the vertex labeled q_k in the graph.

Code optimization by the removal of useless tape symbols, useless states and useless transitions requires reading the input program more than once and hence cannot be implemented in a single-pass compiler. Such a code optimizer can be included only in a multiple-pass compiler. Consequently, the code optimizer phase is not included in the TMDLC.

3 The interpreter

3.1 The design

The IL program generated by the TMDLC contains information just enough to simulate the functioning of the Turing machine. The IL program is free from lexical, syntactic and semantic errors as they have been handled by the TMDLC. This IL program acts as an input to the Turing machine interpreter (TMI). The TMI starts by copying the input string on the tape (Fig. 2). The tape is one-dimensional and infinite in both directions. If the TMI finds a symbol in the input string that is not an element of the input alphabet then it calls the error handler. The TMI sets the given initial state as the current state and the leftmost input symbol as the current tape symbol of the Turing machine. Then the current state and the current tape symbol are matched with the antecedent parts of the transitions. If a match is found, the Turing machine moves to the state mentioned in the consequent part of that transition. The tape symbol mentioned in the consequent part is written back on the tape and the read/write head moves one unit in the direction specified in the consequent part. If a match is not found, then Turing machine is said to halt. The process of matching is carried out iteratively until the Turing machine finds itself in an accepting state. It is assumed that no transition is defined for any accepting state, so the Turing machine will accept the input string whenever it enters an accepting state. The configuration of the Turing machine is displayed in each iteration of the matching procedure using the symbol table passed on by the TMDLC in the IL program.

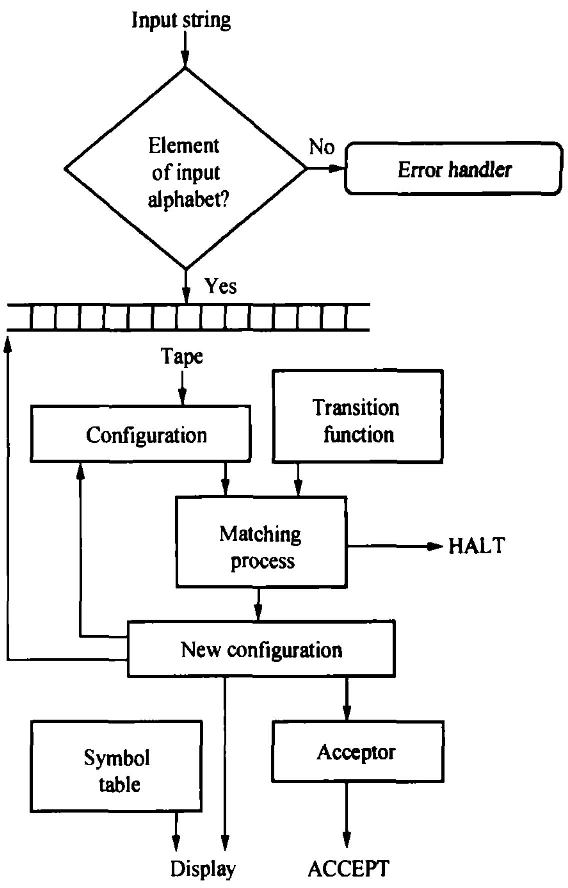

Fig. 2. Block diagram of the TMI.

3.2 The tape

Realization of an infinite tape is not possible in any physical computer. The TMI uses a dynamically allocated array to simulate the tape. The dynamically allocated array is resized on the demand of the Turing machine.

3.3 Representation of the configurations

Let us consider a configuration of a Turing machine, as shown in Fig. 3, with the current state q_i and the current tape symbol a_j. Let the tape content be $a_1a_2\cdots a_{j-1}a_ja_{j+1}\cdots a_{n-1}a_n$, where a_1 and a_n are the leftmost and the rightmost nonblank tape symbols respectively. The TMI represents such a configuration as $a_1a_2\cdots a_{j-1}q_ia_ja_{j+1}\cdots a_{n-1}a_n$. However, if a_{j-1} is the rightmost nonblank symbol on the tape

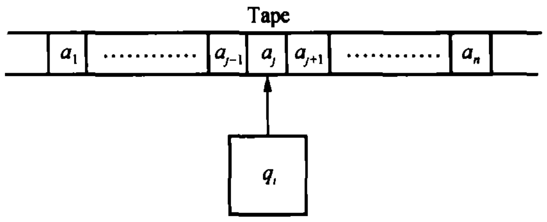

Fig. 3. A configuration of a typical Turing machine.

then the configuration is represented as $a_1a_2\cdots a_{j-1}$ $q_i\square$, where $\square$ stands for the blank tape symbol.

3.4 Integrating the TMDLC and the TMI

Till now, the TMDLC and the TMI have been discussed separately. Fig. 4 illustrates how these two components are used in cascade and what are the different files used by the system. The TMDL program is stored in a .TM file. This file is compiled using the TMDLC. On successful compilation, TMDLC stores the object program in a .ILF file. The .ILF file along with the file containing the input string is provided to the TMI for interpretation. The TMI produces a file containing the final tape content and another file containing the configurations that the Turing machine passes through. The latter also contains the final result, i.e., accept or halt.

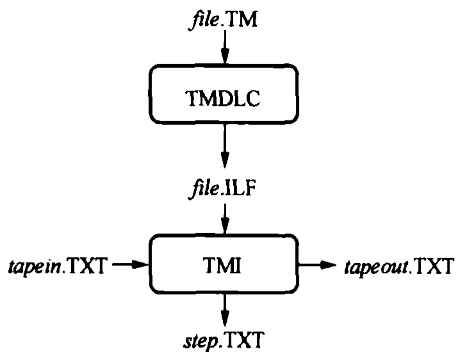

Fig. 4. The system layout.

3.5 Composite Turing machines

A composite Turing machine is a sequence of simple Turing machines each using the output of its immediate predecessor as the input. Such a composite Turing machine can be simulated by interpreting one elementary Turing machine at a time and using its output as the input of the next Turing machine in the sequence.

4 Results and discussion

In this section the performance of the TMI is illustrated for two Turing machines modeled in TMDL.

Turing machine 1

The TMDL program given below defines a TM (6, 2)[5] that accepts the set of all binary strings whose decimal equivalents are multiples of 5.

```
TM = ({q0,q1,q2,q3,q4,qf}, {0,1}, {0,1,_},
    D,q0,_,{qf});
D(q0,0) = (q0,0,R),
D(q0,1) = (q1,1,R),
D(q1,0) = (q2,0,R),
D(q1,1) = (q3,1,R),
D(q2,0) = (q4,0,R),
D(q2,1) = (q0,1,R),
D(q3,0) = (q1,0,R),
D(q3,1) = (q2,1,R),
D(q4,0) = (q3,0,R),
D(q4,1) = (q4,1,R),
D(q0,_) = (qf,_,R);
```

The Turing machine accepts the string 1100100 after moving through the following configurations.

```
q01100100 |- 1q1100100 |- 11q300100 |-
110q10100 |- 1100q2100 |- 11001q000 |-
110010q00 |- 1100100q0_ |- 1100100_qf_ (AC-
CEPT)
```

But, for the input string 1100101 the Turing machine halts after moving through the following configurations.

```
q01100101 |- 1q1100101 |- 11q300101 |-
110q10101 |- 1100q2101 |- 11001q001 |-
110010q01 |- 1100101q1_ (HALT)
```

Turing machine 2

The TMDL program given below defines a TM (3,2)[5] that goes into an infinite loop for an input string aa.

```
TM = ({q0, q1, q2}, {a, b}, {a, b, _}, D, q0, _,
    {q2});
D(q0, a) = (q1, a, R),
D(q1, a) = (q0, a, L),
D(q1, _) = (q2, _, R);
```

The Turing machine keeps oscillating between the configurations q0aa and aq1a and never comes out of the loop. As a result, the TMI enters an infinite loop too.

It is observed that for a given input string and a Turing machine there may arise three situations. Firstly, the Turing machine may accept the string in a finite number of steps. Secondly, the Turing machine may traverse a finite number of steps before halting in a configuration for which no transition is defined. Lastly, the Turing machine may enter in an infinite loop.

5 Conclusions

It is concluded that TMDL is an easy and efficient language to design Turing machines. The salient feature of the language is that a Turing machine can be represented in a way that is very similar to that used in a textbook. The TMI simulates the exact behavior of a Turing machine modeled in TMDL. The technique presented in this paper can be used to study the working of all Turing machines. The TMDL can be extended to allow modeling the different variations of the standard Turing machine like Turing machine with a stay option, Turing machine with a semi-infinite tape, multi-tape Turing machine, multidimensional Turing machine, nondeterministic Turing machine, universal Turing machine, linear bounded automaton and hybrid Turing machine. This will require only minor changes in the TMDLC and the TMI.

References

1 Linz P. An Introduction to Formal Languages and Automata. 4th Ed., Boston: Jones and Bartlett Publishers, 2006
2 Pierce JC, Singletary WE and Vander Mey JE. Tutor—A Turing machine simulator. Information Sciences, 1973, 5: 265—278
3 Aho AV and Ullman JD. Principles of Compiler Design. New Delhi: Narosa Publishing House, 1989
4 Aho AV, Sethi R and Ullman JD. Compilers—Principles, Techniques, and Tools. Delhi: Pearson Education Pte. Ltd., 1999
5 Michel P. Small Turing machines and busy beaver competition. Theoretical Computer Science, 2004, 326(1—3): 25—56