Automaton

Piotr Styczyński (ps386038)

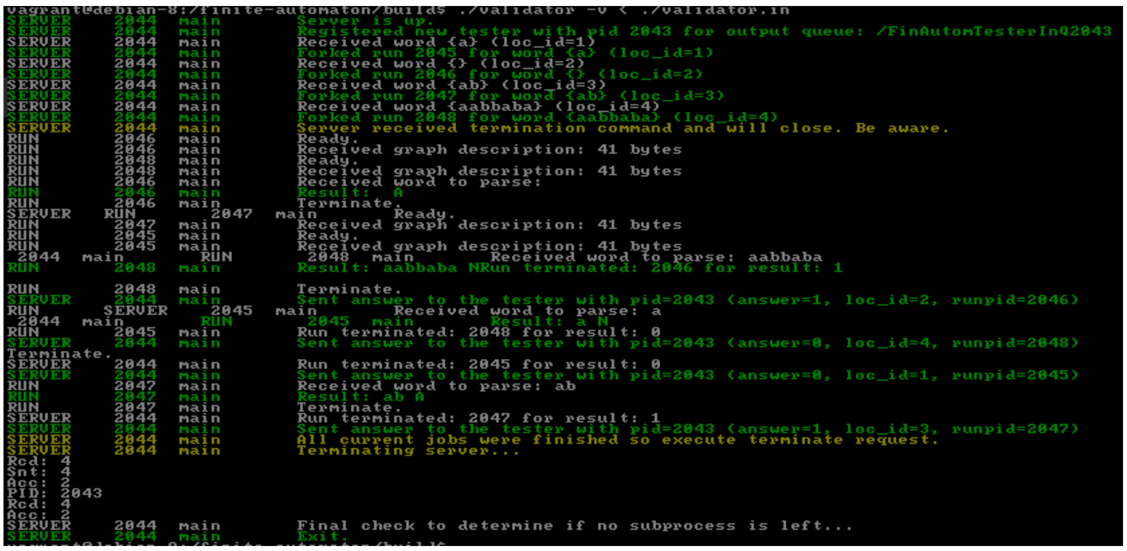
The automaton implementation as a part of the subject on the Warsaw University.
The document index:

Usage

```
./validator [-v] < <automaton_graph_file>
./tester    [-v] < <tester_input_file>
```

Important note:
Note that *-v* switch can be used to enable verbosive debug mode!

Server working with logging enabled:



```
vagrant@debian-8:/finite-automaton/build$ ./validator -v < ./validator.in
SERVER     2044  main       Server is up.
SERVER     2044  main       Registered new tester with pid 2043 for output queue: /FinAutomTesterInQ2043
SERVER     2044  main       Received word {a} (loc_id=1)
SERVER     2044  main       Forked run 2045 for word {a} (loc_id=1)
SERVER     2044  main       Received word {} (loc_id=2)
SERVER     2044  main       Forked run 2046 for word {} (loc_id=2)
SERVER     2044  main       Received word {ab} (loc_id=3)
SERVER     2044  main       Forked run 2047 for word {ab} (loc_id=3)
SERVER     2044  main       Received word {aabbaba} (loc_id=4)
SERVER     2044  main       Forked run 2048 for word {aabbaba} (loc_id=4)
SERVER     2044  main       Server received termination command and will close. Be aware.
RUN        2046  main       Ready.
RUN        2046  main       Received graph description: 41 bytes
RUN        2048  main       Ready.
RUN        2048  main       Received graph description: 41 bytes
RUN        2046  main       Received word to parse:
RUN        2046  main       Result:  A
RUN        2046  main       Terminate.
SERVER   RUN      2047  main       Ready.
RUN        2047  main       Received graph description: 41 bytes
RUN        2045  main       Ready.
RUN        2045  main       Received graph description: 41 bytes
 2044  main       RUN      2048  main       Received word to parse: aabbaba
RUN        2048  main       Result: aabbaba NRun terminated: 2046 for result: 1

RUN        2048  main       Terminate.
SERVER     2044  main       Sent answer to the tester with pid=2043 (answer=1, loc_id=2, runpid=2046)
RUN      SERVER   2045  main       Received word to parse: a
 2044  main       RUN      2045  main       Result: a N
RUN        2045  main       Run terminated: 2048 for result: 0
SERVER     2044  main       Sent answer to the tester with pid=2043 (answer=0, loc_id=4, runpid=2048)
Terminate.
SERVER     2044  main       Run terminated: 2045 for result: 0
SERVER     2044  main       Sent answer to the tester with pid=2043 (answer=0, loc_id=1, runpid=2045)
RUN        2047  main       Received word to parse: ab
RUN        2047  main       Result: ab A
RUN        2047  main       Terminate.
SERVER     2044  main       Run terminated: 2047 for result: 1
SERVER     2044  main       Sent answer to the tester with pid=2043 (answer=1, loc_id=3, runpid=2047)
SERVER     2044  main       All current jobs were finished so execute terminate request.
SERVER     2044  main       Terminating server...
Rcd: 4
Snt: 4
Acc: 2
PID: 2043
Rcd: 4
Acc: 2
SERVER     2044  main       Final check to determine if no subprocess is left...
SERVER     2044  main       Exit.
```

(https://gitlab.com/styczynski/finite-automaton/raw/master/static/screenshot1.png)

There is also a helper to execute testers and server at one time:

```
./autovalidator [-v] <automaton_graph_file> [<tester_input_file1> <tester_input_file2> ...]
```

Which is equivalent to bash set of commands:

```
# Fork
./validator [-v] < <automaton_graph_file> &
./tester    [-v] < <tester_input_file1>   &
./tester    [-v] < <tester_input_file2>   &
...
# Wait
wait
```

Implementation

The implementation of automaton uses utility code that is:

- Partially a code from JNP subject (Warsaw Univeristy) I have written (dynamic/array lists)
- The huge part (written especially for this project) of code that tries to provide tidy API for low level C unix functions.

Quick sources index

Project consists of the following files:

- *array_lists.c* - Implementation of array lists (included in array_lists.h)
- *automaton.h* - Implmentation of automaton data strucutres and machine itself
- *autovalidator.c* - Helper program to launch server (validator) and clients (testers) automatically via one command
- *dynamic_lists.h* - C99 bidirectional linked lists
- *gc.h* - Interface to the GC (more info in GC section)
- *generics.h* - Functions for handling void* (generic) data types
- *hashmap.h* - Generic hashmap based on array/dynamic_lists
- *msg_pipe.h* - Message pipes abstraction for UNIX pipes
- *onexit.h* - Utilites to handle and manage process termination
- *syslog.h* - Easy to use logging interface
- *validator.c* - Automaton server
- *array_lists.h* - Array lists (more or less like C++ std::vector's)
- *automaton_config.h* - Config for automaton server/clients/workers (validator.c)
- *dynamic_lists.c* - Implementation of C99 bidirectional linked lists (included in dynamic_lists.h)
- *fork.h* - Utilities to manage fork/wait behaviour
- *gcinit.h* - Implementation of GC interface (more info in GC section)
- *getline.h* - Implementation of getline in C
- *memalloc.h* - Tools for allocating memory
- *msg_queue.h* - Message queues (mq) abstraction for UNIX message queues
- *run.c* - Automaton server's worker process source code
- *tester.c* - Automaton client source code

General

The automaton is divided into tree separate entities:

- server - which dispatches clients' requests and send them results
- tester - clients which sends words for verification from loaded input
- run - server's worker that is automatically spawned by the server when needed

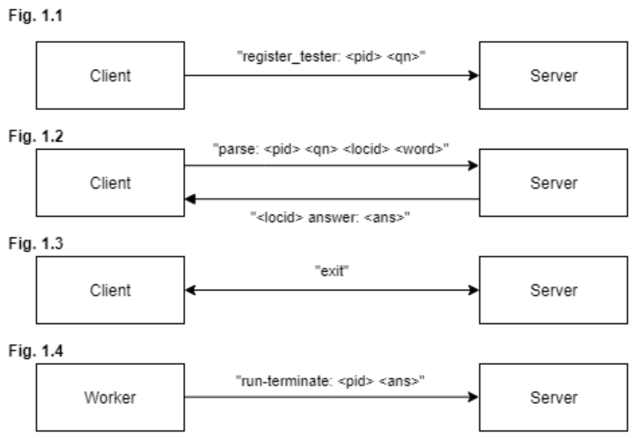
Communication model

Communication between these entities is implemented via several methods:

- Message queues - basic mean of communication
- Pipes - used to sending graph representation (that may be big!)
- Via exec params - used in case of the server which passes the data into the spawned process via its cli parameters

As mentioned in the upper list, the message queues (unix mq) are the base mean of communication.

All the comunicates that are sent through mqueues are described on the diagram below:



(https://gitlab.com/styczynski/finite-automaton/raw/master/static/automaton_com_sch1.png)

The model of application is as follows:

- Registration
  - 2.0 When launched the `tester` process sends "register" event to the server via queue `<server_reg_in>` (fig. 1.1)
    - The message contains `<pid>` - the pid of the tester process
    - The message contains `<qn>` - name of the tester input queue `<tester_ans_in>`
  - 2.1 The server receives the "register" events and saves new session for the client
- Client data loading
  - 3.0 The tester performs data loading
- Parse requests
  - 4.0 The tester send one or more verification requests - "parse" via server queue `<server_req_in>` (fig. 1.2)
    - The request must contain `<pid>` - the pid of the tester process
    - The request must contain `<qn>` - name of the tester input queue `<tester_ans_in>`
    - The request must contain `<locid>` - numerical identificator of the request
    - The request must contain `<word>` - text sequence to be parsed
  - 4.1 The server reads the request and lanuches new worker process
    - The worker process receives `<worker_graph_in>` pipe id (worker graph input pipe)
    - The worker process receives `<word>` - word to be parsed
  - 4.2 The worker receives automaton graph data using `<worker_graph_in>`
- Worker internal calculations
  - 5.0 The worker performs internal caluclations
- Submiting worker results
  - 6.0 After the worker has got its answer it sends "run-termiante" event to the server via server queue `<server_ans_in>`
    - The answer contains `<pid>` - the pid of the worker process
    - The answer contains `<ans>` - the generated answer
  - 6.1 The server receives worker termination events and sends answer back to the client (via `<tester_ans_in>`)
    - The answer contains previously sent `<locid>`
    - The answer contains the server answer `<ans>`
- Exitting
  - 7.0 Optionally the client may want to stop the server with "exit" (fig. 1.3) message
  - 7.1 When server receives the exit message it's broadcasted to all opened client sessions
    - The broadcast happens after all unfinished server processes will terminate
    - After registering exit request no other "parse" request will be executed
  - 7.2 When client receives the exit message then it terminates

The server opens the following mqueues/pipes:

- `<server_reg_in>` - fixed queue name; used for incoming tester registration events
- `<server_req_in>` - fixed queue name; used for incoming tester parse requests
- `<server_ans_in>` - fixed queue name; used for incoming worker termination events
- Per each worker opened session:
    - `<worker_graph_in>` - pipe to send the automaton graph data to the process
- Per each tester opened session:
    - `<tester_ans_in>` - used for sending answers to the tester (it's name is passed via "register" and "parse")

The client opens the following mqueues/pipes:

- `<tester_ans_in>` - queue used for incoming answers
- `<server_reg_in>` - queue used to register in the server
- `<server_req_in>` - queue to send the parse requests

The worker opens the following mqueues/pipes:

- `<worker_graph_in>` - pipe to receive graph data from the server
- `<server_ans_in>` - queue to transmit calculated answers back to the server

The point **4.1** is implemented via passing-by-args method.
The worker is launched with the command:

```
./run <worker_graph_in> <word>
```

The server side session-independent queues has got fixed names.
And the tester has full flexibity to generate any input queue name `<tester_ans_in>`.

Sessions

When "register" event is received by the server (it has priority over any other client request), the new session for the client is created and associated with it's pid passed in the message.

The "register" message is optional and sessions are opened automatically on normal "parse" events.

However "register" event thanks to its priority will ensure that all clients launched before the server, will register correctly.
The correct registration is crusial when it comes to the "exit" (point *7.0*) broadcasting (as we want to close all the clients).

The sessions are captured in the server inside HashMap indexed by process (client/worker) `pid`.

Locids

The `<locid>` is abbreviation for local id.

The local id can be freely generated by the client.
It will be associated with the parse request.
The server identifies the worker answer by the locid value that is captured inside session data for the worker.

The communication model assumes that the client saves the information that the local id is associated with some word, because response from the server contains only local id and not the word itself.

The best approach in this situation is to generate local ids sequentially and save the words in array.
Below I attach some C-like pseudocode:

```c
char* words[100];

int locid = 0;
for(int i=0;i<100;++i) {
    words[locid] = load_input_word();
    send_request(locid, words[locid]);
    ++locid;
}

int result = 0;
while(1) {
    int locid = wait_for_answer(&result);
    printf("asnwer for word %s is %d", words[locid], result);
}
```

Inter-process worker communication

The `acceptAsync` methods forks itself when needed (this is determined heuristically - for more details see `automaton.h`)

The communication between self-forked worker process is done via pipes.
The parent opens pipes for back-communication.
The child process calculates results and send the back via the parent pipe.

Errors

The system functions inside utility functions are checked agains failures.
In most cases the function will trigger `syserr` which terminates program with -1 exit code immidiately.

In case when run forked itself and some of the children returned non-zero exit code or terminated abnormally then the process kills itself with -1 exit code.

In case when any worker process terminated with non-zero exit code, the server kills itself trying to broadcast "exit" to all connected clients and return -1 exit code.

In case of server abnormal termination it will always try to broadcast the "exit" message if it's possible.

Memory leaks

The application is designed to free all used resources.

In case of abnormal termination, exit(...) does not deallocate memory.
It's critical (specified in the study task) to prevent any memory leaks (also that with exit, even if OS handles this case for us).

That's why I wrote simple GC that uses HashMap to register all allocs/frees and free the memory on termination using `atexit` .
The GC registers only allocations done via `memalloc.h` functions and macros.

Assumptions

The following assumptions were done during implementation:

- Mamium line length of any input is `LINE_BUF_SIZE`
- Maximum file input length (of graph representation) is `FILE_BUF_SIZE` and it fits into memory
- Maximum number of states is `MAX_Q`
- Letters are letter of ascii code `{'a', 'b', ..., 'a'+MAX_Q-1}`

This values as others settings can be changed in `automaton_config.h` .

The detailed code explenation is done in each file.
Please refer to `validator.c` , `tester.c` and `run.c` to obtain more implementation detailed info.

Abstract

Definitions

> **A finite automaton** -- a simple state machine which accepts or rejects words.
>
> **A word** is any finite sequence of letters from an alphabet.
>
> **An alphabet** is any finite set.

If *w* is a word , then by |*w*| we denote its length, and by *w[i]*, *0 <= i <* |*w*|, its *i-th* letter.
Moreover, the symbol ε denotes an unique empty word, i.e. a word of length 0.

Formal model

Formally, an automaton *A* is a tuple (*L, E, U, F, qI, T*), where:

- *L* is an alphabet,
- *E* is a set of existential states,
- *U* is a set of universal states,
- Q ≝ E ∪ *U* is the set of all states of *A*,
- *F∈Q* is the set of final states,
- *qI* is the unique initial state,
- *T: Q × L → P(Q)* is the transition function.

Intuitively, an automaton recognises a word in the following way: the automaton starts in the initial state and reads the word, letter by letter. When the automaton is in state *q* and sees a letter *a* then it '*chooses*' the next state from the set *T(q,a)* and proceeds. If *q* is universal then every such choice has to lead to acceptance, if *q* is existential then only one choice has to lead to acceptance. Acceptance, in this context, requires that the above process after reading the last letter of the word ends up in a final state.

More formally:

the above process can be described as a tree.
A run of the automaton run on word *w* of length *d* is a tree of height *d+1*.
(For instance, a run on the epty word has unique node -- the root -- at the depth 0.)

The root ε of the run is labelled *run[ε] = qI* and every node *u* in the tree satisfies the following:

If node *u* is at depth *i*, is labelled *q* and:

- T(q, w[i]) is empty: then u is a leaf;
- T(q, w[i]) isn't empty and the state q is existential: then u has exactly one child labelled with a state belonging to the set T(q, w[i]);
- T(q, w[i]) isn't empty and q is universal: then node u has exactly one p-labelled child for every state p ∈ T(q, w[i]) .

We say that a run is accepting if the leafs' labels on depth *d* are final states and the labels of the remaining leafs are universal states.

A word is accepted by an automaton if there is an accepting run.
If there is none then the automaton rejects the word.

The acceptance of a word can also be described as a recursive function.
An automaton accepts a word *w* if the following `function bool accept(string w, string r)`
returns `true` when called as `accept(w,qI)` .

```
// This function accepts the word @w continuing the process from the last node of the path @r
// belonging to some run of the automaton
bool accept(string w, string r):

  d := |r|-1;

  if (d ≥ |w|)
    // We have processed every letter in @w; accept if the last state is accepting
    return (r[d] ∈ F);

  if (r[d] ∈ E)
    // The last state is existential so accept if there
    // is an accepting continuation of the run @r
    return (∃q∈T(r[d],w[d]) accept(w,rq))

  // Otherwise, the last state is universal (r[d] ∈ U)
  // and we have to accept every possible continuation
  return (∀q∈T(r[d],w[d]) accept(w,rq))
```

Note:

Variables *w*, *r* are words and *rq* is the conctatenation of the word *r* and the letter (also a state) *q*.
Intuitively, *r* can be seen as a path in a run.

Specification

> A sequence [wyr] denotes that the string wyr repeats a finite (greater than or equal to 0) number of times.

Proces validator

The validator is a server accepting, or rejecting, words. A word is accepted if it is accepted by
specified automaton. The program validator begins by reading from the standard input
the description of the automaton and then in an infinite loop waits for the words to verify.

When a word is received he runs the program run which validates the word and vaidator waits
for an another word or response from the program run. Afer receiving a message from one of the
run processes validator forwords the message to the adequate tester.

When a tester send an unique stop word ! the server stops , i.e. he does not accept new words,
collects the responses from the run processes, forwards the answers, writes a report on stdout, and, finally, terminates.

The validator report consist in three lines describing the numbers of received queries, sent answers and accepted words:

```
    Rcd: x\n
    Snt: y\n
    Acc: z\n
```

where x,y,z respectively are the numbers of received queries, sent answers and accepted words; and a sequence of summaries of the interactions
with the programs tester from which validator received at least one query.

A summary for a tester with PID pid consists in:

```
    [PID: pid\n
    Rcd: y\n
    Acc: z\n]
```

where *pid*, *y*, *z* respectively are:

- the process' pid,
- the number of messages received from this process
- the number of acceped words sent by this process.

Proces run

Program run receives from validator a word to verify and the description of the automaton.
Then he begins the verification. When the verification stops, the process sends a message to the server and terminates.

Proces tester

Queries in a form of words are sent by programs tester.

A program tester in an infinite loop read words form the standard input stream and forwards them to the server.
Every word is written in a single line and the line feed symbol \n does not belong to any of the words.
When the tester receives an answer from the server,
he writes on the standard output stream the word he resecived answer for and the decision A if the word
was accepted and N if not.
The tester terminates when the server terminates of when tester receives the EOF symbol,
i.e. the end of file symbol. When the tester terminates it sends no new queries,
waits for the remaining answers from the server, and writes on the standard output a report.

A Report of a tester consist of three lines

```
    Snt: x\n
    Rcd: y\n
    Acc: z\n
```

where *x*, *y*, *z* respectively are the numbers of:

- queries,
- received answers,
- accepted words sent by this process

Input specification

Program validator

Input of a validator consist in the automaton description:

```
  N A Q U F\n
  q\n
  [q]\n
  [q a r [p]\n]
```

Where:

- *N* is the number of lines of the input
- *A* is the size of the alphabet: the alphabet is the set *{a,...,x}*, where *'x'-'a' = A-1*
- *Q* is the number of states: the states are the set *{0,...,Q-1}*
- *U* is the number of universal states: universal states = *{0, .., U-1}*, existential states = *{U, .., Q-1}*
- *F* is the number of final states
- *q*, *r*, *p* denotes some states
- *a* is a letter of the alphabet

The first line gives the cardinalities of the sets describing the automaton.
The second line gives the initial stare,
the third line is the list of final states.

The remaining lines, of form: *q a r [p]\n* , encode the transition function in the following way:
if there is a line *q a p1 p2 ... pk* then *T(q,a) = {p1, p2, ..., pk }*
If for any pair *(q,a)* there is no line in the encoding then one assumes that *T(q,a) = ∅*

One can assume that: *0 <= F,U <= Q < 100, 0 <= A <= 'z'-'a'*

Program tester

A tester on the standard input stream receives the following string.

```
  [[a-z]\n]
  [!\n]
  [[a-z]\n]
```

Where

- *a-z* are the alphabet letters;
- *!* is the unique terminate the server symbol.

One can assume that the length of the supplied words will not exceed the constant *MAXLEN = 1000*.