December 1987 LIDS-P-1728

An Event Driven Job Shop Simulator

Sheldon X.C. Lou
Laboratory for Information and Decision Systems

December 25, 1987

Abstract

The simulation software described in this document simulates a job shop with multiple machines/work stations and parts. The recipes for parts can be assigned by the user. The simulator features complete separation of control and simulation, which is particularly pertinent for research and development of new scheduling algorithms. Also, due to the object oriented programming technique, the program is well structured. Therefore, further expansion is easy to implement.

1 INTRODUCTION

There are several reasons why we need to write a simulator. Firstly, it is usually very inconvenient to implement user defined control algorithms and statistic evaluation techniques. Unfortunately, these may very well be the primary requirements for research and development of scheduling algorithms. In the simulator described in this report, the simulation engine and the control rules are placed in different modules and the appropriate interface is provided, the user will find implementing his/her own control algorithm fairly easy. Further, due to the object oriented programming, modularity design and the list manipulation functions (see Section ??) furnished by the program, tailoring the program to fit into user's requirement becomes simple. Lastly, the program is written in C language. Although the program is implemented in UNIX operating system, due the portability of C, it is trivial to transfer it to other operating systems.

In the next section, we first describe the data structure of the program. Then we explain the mechanism of the simulator.

2 DATA STRUCTURE

The data structure of the program is depicted in Fig. 1.

There are several lists kept in the system. First is the Work–station–list which contains the work stations[1] in the job shop. There are two lists (or queues) related to each work station. The first is the *Buffer–Queue* which represents the queue formed by the loads placed in the buffer before the work station. The load at the top of this queue will be loaded first. The function of the control algorithm is then to sort this queue according to appropriate rules. The second queue is the *Process–Queue* which is the loads being processed by the work station.

The parts being processed in the job shop are contained in the *Part-List.* Note, by *Part* we refer to the entity of all identical loads (although the size may vary). Associated to each part is the *Recipe* of this part which in turn contains several job steps. Another important queue/list in the system is the *Event–Queue* which consists of future events sorted according to the time they will happen.

Among them, the Buffer–Queue, Process–Queue, and the Event– Queue are dynamic. They are constantly updated by different rules. For instance, when one work station finishes a load, this load will be removed from the Process–Queue and put into the Buffer–Queue of the next work station. If one load is being loaded into a work station, this load will be removed from the Buffer–Queue of this work station and put into its Process–Queue. The Buffer–Queue will be sorted when a load should be loaded according to the control rule being used. When an event, such as work station failure, repair, part loading, is created, it is placed into the Event–Queue according to its time to happen and the event which just happened will be deleted from the Event–Queue immediately.

The connections among entities (such as work stations, loads, job steps) and lists can also be seen from Fig. 1. We have just mentioned the relations among different lists. Each load points to certain part it belongs to and

[1]For our research purpose, we use work station instead of the machine as the basic unit. But it is without any difficult to use the latter.

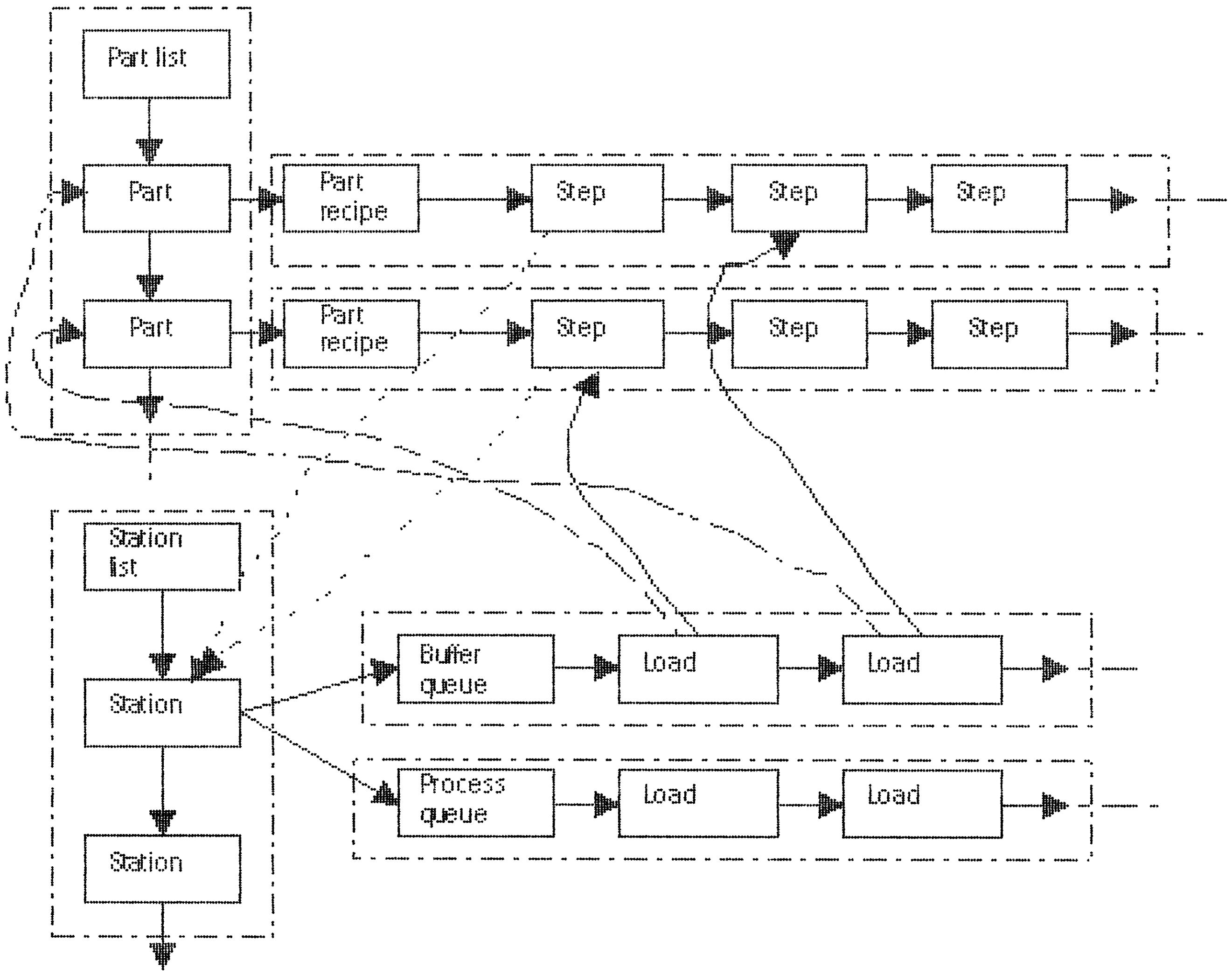

Figure 1: Data structure of the simulator

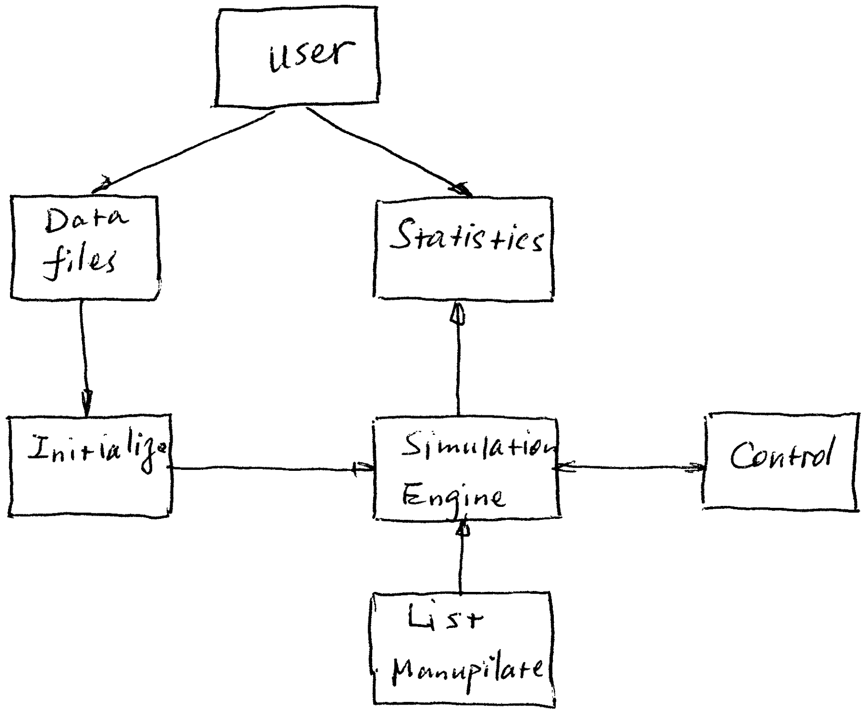

Figure 2: Modules of the simulator

the job step it currently resides. The latter is updated constantly when the load moves along its processing route defined by the part recipe. Each job step points to the work station it requires.

3 MODULES AND THEIR FUNCTIONS

The simulator contains four modules as shown in Fig. 2.

The first one is the Initialization module which interacts with the data files (provided by the user) to establish the data structure as shown in Fig. 1 and the initial conditions (including the initial loads at the different work stations) of the system.

The main body of the simulator is the Simulation Engine module and the Controller module. The function of the Simulation Engine is to take proper actions according to the event at the top of the event queue. There are five types of events defined in this program. The first one is STATION–FINISH which represents the events when a work station finishes a load.

The program then remove this load (the one at the top of the process queue) from the process queue of this work station and put it into the buffer queue of the work station of the next job step according to the recipe of this load. If this buffer queue is empty, then the program immediately fires another event—CHECK–FIRST–MACHINE (see next) in order to see if an immediate loading is appropriate.

The second kind of events is FIRST–MACHINE–FINISH which represents the events when the first machine in a work station finishes processing. The program then checks the appropriate control rules specified by the user and the work station condition (up or down) to see if a load in the buffer queue of this work station should be loaded. If the answer is yes, it calls the CONTROL module and tries to determine which and how many parts should be loaded considering also the capacity of the work station. It moves those loads that should be loaded to the Process–Queue of the work station.

The next one is CHECK–FIRST–MACHINE which is fired when a load is moved to an empty Buffer–Queue. It first checks if the first machine of the work station is idle. If it is, then the program does the same thing as that of the FIRST–MACHINE–FINISH event does.

The fourth and fifth event categories are STATION–FAILURE and STATION–REPAIR. The time between those two events are randomly generated by a random number generator.

The Control module contains several sub–modules, each represents a control strategy. For example, the TWO–BOUNDARY control loads parts according to the station's BUFFER–HEDGING–POINT and SURPLUS–HEDGING–POINT set by the user. For re-entrant processes, it gives higher priority to later processes. The UNIFORM-LOADING control adds equal amount of parts before the first work station at equally spaced intervals. For example, the user may require that 100 loads of certain parts should be put in the buffer of the first work station at every 100 hours.

The STATISTICS module computes the total costs of different control strategies.

LIST AND ITS MANIPULATION FUNCTIONS

In this program, lists are constructed as seriesly connected building blocks called CONS. The head of each CONS points to an element (such as a work station of a work station list) of the list. The tail of the last CONS points to NULL (or the zero integer in this program). We provide number of list manipulation functions as described in the following.

- first(list), which returns the first element of the list.
- rest(list), which returns the rest of the list(the list starting from the second element).
- last(list), which returns the last element of the list.
- position(element, list), which returns the position of the element in the list.
- insert(element1, element2, list), which inserts element1 before element2 in the list.

HEADER FILE
The following is the header file of this program.

```
/*********************************************************
*       The header file for the simulation program *
*********************************************************/

#define LENGTH  20
#define TRUE    1
#define FALSE   0
#define DOWN    0
#define IDLE    1
#define BUSY    2
#define FIFO    0
#define TWO_B   1
#define UNI_LOAD 2
#define NUMBER_OF_PART 1
#define STFIN   1
#define FSTMACFIN  2
```

```
#define FAIL    3
#define REPR    4
#define CHECKLOAD  5

typedef char    NAME[LENGTH];
typedef char    LINE[100];
typedef int     BOOLEAN;
typedef long    TIME;
typedef char *  P_CHAR;

/*******************************************************
*       CONS is a building unit for list              *
*                                                     *
*******************************************************/

typedef struct cons {
        int                     head;
        struct  cons *          tail;
        }  CONS;

typedef CONS *   P_CONS;



/*******************************************************
*       Work station                                  *
*                                                     *
*******************************************************/

typedef struct station {
        int       id;
        NAME      name;
        CONS *    buffer;         /*parts in buffer*/
        CONS *    process_q;      /*parts in process*/
        int       status;         /*DOWN: station is down*/
                                  /*IDLE: up, 1st mac idle*/
                                  /*BUSY: up, 1st mac busy*/
        long     seed;           /*seed to generate up & down*/
        TIME     ave_up_time, ave_down_time,
                 time_to_down, time_to_up,
                 last_start;     /*time a part had loaded */
TIME     first_proc_time;/*proc time of 1st mac*/
        TIME     proc_time; /*proc time of station*/
        int      capacity;  /*maximum loading cap*/
        int      load_quant; /*quantity of parts loaded*/
        BOOLEAN  load_permit; /*if further loading permitted*/
      }       STATION;

typedef STATION *       P_STATION;

/*****************************************************
*       Job step                                     *
*                                                    *
*****************************************************/

typedef struct step {
        NAME        step_name;       /*name of job step*/
        NAME        station_name;
        STATION *   station;        /*station being used*/
        long        wip;        /*integral of wip over time*/
        float       cost;       /*integtal of cost over time*/
        float       weight;     /*weight for certain part*/
        TIME        proc_time;  /*processing time*/
        TIME        first_proc_time;/*proc. time of 1st mac*/
        float       yield;          /*yield of this step*/
        int         wip_thres;      /*threshold for wip*/
        int         surplus_thres;  /*threshold for surplus*/
        }       STEP;

typedef STEP *          P_STEP;

/*****************************************************
*       Part: the part the job shop are producing    *
*                                                    *
```

```
*****************************************************/

typedef struct part {
        int      part_id;
        NAME     part_name;
        NAME     part_recipe_name;  /*recipe the part uses*/
        CONS *   part_recipe;
        int      target_prod_rate;  /*target production rate*/
        }       PART;

typedef PART *          P_PART;

/*****************************************************
*****************************************************/

typedef struct load {
        NAME    load_name;  /*load id number*/
        STEP *  load_step;  /*current job step*/
        PART *  load_part;  /*the part this load belongs to*/
        TIME    finish_time;/*time proc. finishes*/
        int     load_size;  /*number of lots contained*/
        }       LOAD;

typedef LOAD *          P_LOAD;


/*****************************************************
*       Event                                        *
*****************************************************/

typedef struct event  {
        TIME    time;   /*time to happen*/
        int     flag;   /*STFIN:finishing process at WS*/
                 /*FSTMACFIN:finishing process at first mac*/
                 /*FAIL:WS failure*/
                 /*REPR:WS repair*/
                 /*5:*/
        P_STATION   station;        /* related station*/
        P_LOAD      load;/*if finishing process, related load*/
        }       EVENT;

typedef EVENT *         P_EVENT;
```

DATA INPUT FILES
There are four data input files.

1. Gene.dat contains information about simulation time horizon, dispatching rules used, and how much intermediate information should be printed.
2. Load.dat defines the information about the initial loads, such as their part names, number of loads, their scheduled finish times, and the size of each load.
3. Part.dat provides the recipe for each part. It also gives the hedging points required by Two–Boundary control rules.
4. Station.dat shows the stations used in the system, their average up time, down time, processing time and their capacities.

Users of this program should first set up those four data files according to the manufacturing system they are simulating before they start the simulator.