Testing Software Product Lines: A Case Study

S. Tawfig[1*], A. Mili[2]
[1] SUST, Khartoum, Sudan.
[2] NJIT, Newark NJ USA.

* Corresponding author. Tel.: +9665332256; email: s.tawfig2006@gmail.com, sara_tawfig@hotmail.com
Manuscript submitted October 16, 2014; accepted December 17, 2014.
doi: 10.17706/ijcee.2015.v7.870

Abstract: Testing traditional software products follow a systematic process that includes: test data generation; oracle specification and design; test execution and test outcome collection; test outcome analysis and product remediation. This simple lifecycle is challenged in product line engineering because it is not possible in domain engineering, and it is not practical in application engineering. In domain engineering, we have no finished executable product, nor do we have a complete requirements specification to use as the basis for oracle design. In application engineering, it is not practical to apply the testing process to each new application as if it were developed from scratch; rather it would be beneficial to capitalize on the verification and validation that we may have done in domain engineering, and on the testing that we may have carried out on other applications within the same product line. In this paper, we explore, on the basis of a sample example, possible lifecycles for product line software testing that include domain engineering and application engineering steps.

Key words: Application engineering, domain engineering, product line engineering, software testing.

1. Position of the Problem

Testing traditional software products follows a systematic process that includes:

- Test data generation, whereby we define coverage criteria, then generate test data to meet the criteria.
- Oracle specification and design, whereby we decide what functional properties of the software product we want to test, then generate an oracle that can determine, for each test instance, whether the software product behaved according to the prescribed behavior.
- Test execution and test outcome collection, whereby we run the software product on all the test data that was identified in the first step, and check it against the test oracle put forth in the second step. As the test proceeds, we record all relevant outcomes, such as which executions were successful or unsuccessful.
- Test outcome analysis and product remediation, whereby we analyze the test outcomes collected in the previous phase and draw conclusions depending on the purpose of the test: these conclusions may be an estimate of product reliability, a diagnosis of possible product faults, a ruling on the certification of the product, etc.

This neat step-by-step process is impractical when we are dealing with product lines, because it is not feasible in the domain engineering phase, and it is not efficient in the application engineering phase [1].

- Indeed, in the domain engineering phase, two obstacles interfere with deployment of a test process: First, we do not necessarily have executable software products that we can test, as we do in traditional

software lifecycles; Second, we do not have a formal specification that describes in detail the expected function of the software product, and would enable us to derive a test oracle.

- At the application engineering phase, we do have an executable software product, namely the finished application, and we do have a formal product specification that can be used to derive a test oracle. The problem that arises in this phase is one of efficiency: Given that the different applications of the domain have broad attributes in common (referred to as commonalities), how can we exploit these to generate test data once, perhaps at domain engineering time, and use it (with some application-specific variations) at application engineering time? Also, given the different application of the domain have much software assets in common (with possibly some variations), how can we capitalize on this commonality to target the test of each application to application-specific functionalities, and let the common functional attributes be tested only once for the whole product line, rather than needlessly repeated for each individual application?

Rather than speculate on these questions, we propose, in this paper, to investigate them on a concrete example, from which we try to draw some general lessons and conclusions. In the next section, we introduce our sample product line, by discussing its general requirements, as well as its variabilities. In Section 3, we present possible reference architecture and a set of reusable/ adaptable assets that enable us to support the product line. In Section 4 and Section 5 we discuss the software testing activities that can be carried out on this product line at, respectively, domain engineering then application engineering. In Section 6 we explore the lessons and conclusions that can be drawn from our experiment with the proposed product line.

2. A Sample Product Line Specification

We wish to develop a product line of applications that simulate the behavior of waiting queues at service stations [2]. These may represent customers standing in line at checkout counters at a store, or travelers standing at airline check-in counters at an airport, or arriving passengers standing at immigration stations in an international arrivals terminal, or postal customers standing in line for service at a post office, or processes being queued at a shared resource allocation post, etc. The purpose of the applications in this product line is to enable managers to simulate various queuing and servicing policies, and analyze their performance in terms of waiting time, fairness, throughput, etc.

Among the commonalities that we envision between all the applications of this product line, we cite the following:

- *The Input Data to the Simulation*. The user must enter the following information:

1) The duration of the simulation, as a function of a virtual unit of time (e.g. the minute for simulations of customers, the millisecond for simulations of processor allocation, etc).
2) The arrival rate, i.e. the average length of time between two successive arrivals, expressed in the unit time selected above. If there are more than one category of customers, then a rate for each category.
3) The service rate, i.e. the average length of service time required by each customer. If there are more than one category of customers, then a rate for each category.

- *The Format of the Output Data*. The user may be interested to collect a variety of statistics pertaining to the simulation. The set of possible functions he may be interested in varies from one application to another, and is determined at application engineering time, as we discuss below.
- *A Standard Record Structure for Customers*. We could make the customer record structure a variability, but for the sake of simplicity we choose to adopt a generic structure that will represent most of the relevant data, such as some identification of the customer, his time of arrival, his category (if there are more than one), his requested service time, his priority (if queuing is based on priority), and his time

of departure.

- *An Illustration of the Simulation*. The simulation may be illustrated at run time by showing the evolution of the waiting queues and the service stations throughout the simulation process.

Among the variabilities between applications in this product line, we cite:

- *Topology of Service Stations*. An application may have a single service station or several service stations; if there are several service stations, they may be interchangeable (offering the same service) or not (offering different services, for example: first class passengers vs business class passengers vs coach passengers at an airline check-in area; or self-check-out counters vs attended check-out counters vs check-out counters for small orders at a store's cash registers; or citizens vs permanent residents vs visitors at immigration posts at an international arrivals hall of an airport). Another dimension of variability is whether a service station, if it is available, may serve customers from a different class, if such customers are not being served by their corresponding service station (for example, if a first class check-in station is available and there are coach passengers in the coach waiting queue, we may want to have them served at the first class station).
- *Service Time*. The service time of a customer may be fixed (the same for all customers) or it may be variable (most typical, in practice). If it is variable, its length may depend solely on the customer (some customers need more service than others) or it may depend on the customer and on the service station (combining the customer needs with the productivity/ efficiency of the service station attendant). If the service time is variable, it may be subject to a maximum allocation (as is the case in round-robin allocation of CPU cycles to competing processes in an operating system).
- *Topology of Queues*. Given a configuration of service stations (one or many, interchangeable or distinct, with or without cross servicing), we may have one queue per service station, or one queue per category of service stations.
- *Arrival Distribution*. We can imagine a number of probability laws that govern the arrival of new customers. Possible options may include:

1) A uniform probability distribution.
2) A Markovian probability distribution.
3) A Poisson probability distribution.

- *Queuing Policy*. This policy deals with two questions: given an arriving customer, what queue do we put him in, and where in the queue do we place him. In terms of the first question, options include:

1) Each category of customers is assigned to a particular type of queue, if there is one queue per category.
2) Each category of customers is assigned to the shortest queue that corresponds to his category, if there is more than one queue per category.
3) Customers are randomly assigned to a queue that corresponds to their category.

As for how to place each customer in the selected queue, we consider two options: a FIFO policy or a priority based policy.

- *Dispatching Policy*. Whereas queuing policy deals with where to place an incoming customer in the queue system, the dispatching policy deals with which customers to pick for service whenever a service station becomes available. The simplest situation is to have a queue associated to each service station, and not to allow cross-queue transfers. Other options include the situation where several interchangeable service stations take their customers from a shared queue and the situation where an idle service station can take customers from the queue of another service station.
- *Measurements*. Measurements include any combination of:

1) Average, median, minimum or maximum waiting time, i.e. time spent in waiting queues.
2) Average, median, minimum or maximum sojourn time, i.e. time spent in the system overall.

3) Fairness, i.e. the extent to which waiting time is proportional to requested service time.
4) Occupancy rate of the service stations, i.e. the extent to which service stations were busy.
5) Throughput of the service stations, i.e. the number of customers serviced per unit of time.
6) Total duration of the simulation (if the simulation is allowed to proceed until all customers are serviced).
7) Total number of customers serviced.

- *Wrap Up Policy*. When the user of an application determines the length of the experiment, a number of decisions must be made as to how the simulation winds down:

1) The simulation is stopped abruptly when the selected time elapses; then all remaining customers are flushed out, possibly taking their statistics.
2) When the time of the simulation is exhausted, no new customers are generated but the simulation continues until all the current customers have been serviced and have exited the system.

In the next section, we consider a possible reference architecture for applications in this domain, and then we implement some reusable/adaptable components of this architecture, and outline how such components are composed to produce an application.

3. A Sample Implementation

3.1. Domain Modeling

We have to make provisions for all possible configurations of the service stations and corresponding queues, namely: variable number of service stations, variable number of queues, variable number of service station types, various mappings from queues to service stations, etc. In order to cater for all possible configurations, we resolve to introduce a basic building block, which we call the queue-station block; each such a block is made up of a number of interchangeable service stations, and a single queue feeding customers to these stations. We represent such a block by the symbol QS(*n*), where n is the number of service stations, and QS stands for Queue-Service Station. We leave it to the reader to check that all possible queue/ station configurations can be implemented by a set of such blocks, with varying values of n. For example, if we want to simulate the situation of an airline check in counter that has 2 stations for first class, 3 stations for business class, and 5 stations for coach, we write (in the style of a type-variable declaration):

- QS(2) firstclass; QS(3) businessclass; QS(5) coach.

In addition to specifying the number of service stations in a QS component, we may want to also specify the queuing policy; if we want the first class and business class queues to adopt a FIFO policy, but want to adopt a priority-based policy for coach queues (e.g. award some privileges to frequent flyers who still fly coach), we may write:

- QS(FIFO, 2) firstclass; QS(FIFO, 3) businessclas; QS(PRIORITY, 5).

If, for example, we want to simulate the situation of waiting queues at a gas station, where each pump has its own queue of cars and cars are served by order of arrival, then we write:

- QS(FIFO, 1) pump1, pump2, pump3, pump4, pump5.

In addition to specifying the configuration of queue/station sets, we may also want to specify policies pertaining to how some service station may serve the queues of other service stations; for example, in an airline check-in counter, it is common for first class station to serve coach passengers is the station is free and the coach queue is not empty. To this effect, we use the feature CrossStation, and we consider the following options to this feature:

- CrossStation(NONE): no such a possibility is available.
- CrossStation(S,Q): specifies the station that offers the service, and the queue to which the service is offered.

For example, in the case of an airline check-in counter, we may write:

- CrossStation(firstclass, businessclass);
- CrossStation(firstclass, coach);
- CrossStation(businessclass, coach).

To feed customers to the simulation, we create a component called Arrivals, which generates customers according to the arrival rate provided by the user at run-time. This component implements the arrivals distribution of the application, and is responsible for the implementation of the queuing policy, at least as far as dispatching arriving customers to QS stations, according to their category, or to some other criterion. We assume that the Arrivals component takes two parameters:

- The law of probability that determines the arrival of new customers at each unit of clock time: UNI (for uniform), MARKOV (for Markov), or POISSON (for Poisson).
- The rule that determines where each new arriving customer is queued: We assume that we have three options, as discussed above, namely: CAT (by category), SHORT (to shortest queue), ANY (random assignment to queues).

In addition to specifying where incoming customers are placed, we may also want to specify whether the assignment of customers to queues is permanent (until the customer is served) or whether a customer may jump from one queue to another. We use the feature CrossQueue to this effect, and we consider the following options to this feature:

- CrossQueue(NONE): The assignment of customers to queues is permanent.
- CrossQueue(FRONT, *n*), where *n* is a natural number: if a queue is of length *n* or greater and another queue is empty, the front of the first queue is sent to the empty queue.
- CrossQueue(BACK, *n*), where *n* is a natural number: if a queue Q1 is longer than a queue Q2 by *n* elements or more, then the back of queue Q1 is moved to the back of queue Q2.

We assume that CrosQueue transfers take place only within QS sets of the same category.

Also, to collect statistics pertaining to the simulation, we create a component called Statistics, that is called by the QS stations whenever a customer is about to leave the system after being serviced; it may also be called by the QS stations at each iteration, if the user is interested in measuring the rate of occupancy of service stations. This component collects data about individual customers, then computes simulation-wide statistics at the end of the simulation and posts it to the user. We assume that this component lists as parameters all the statistics that are selected at application engineering time, including the following:

- Waiting Time (parameter: WT), including minimum, maximum, average, median.
- System Response Time (parameter: SRT), i.e. time spent by customers in the system, including minimum, maximum, average, median.
- Occupancy rate for each station (parameter: OR).
- Maximum queue length for each queue (parameter: MQL).
- Throughput of the system, i.e. number of customers served per unit of time (parameter: TP).
- System Fairness (parameter: FAIR).

It is possible to envision an Application Modeling Language, in the style advocated by Weiss and Lai [3], that we use to characterize each application of this domain. In addition to all the details specified above, the language may include an indication of whether the simulation ends abruptly when the simulation time runs out, or whether winds down smoothly until all residual customers have been serviced and leave the system. This can be written as:

- WrapUp (ABRUPT), or
- WrapUp (SMOOTH).

Hence, for example, if we wanted to specify the simulation of waiting queues in a gas station, we would

write:

```
Simulation gasStation
    {
      QS(FIFO, 1)    pump1, pump2, pump3, pump4; // four gas pumps
      CrossStation (NONE); // each pump services its own queue
      Arrivals (MARKOV, ANY); // arrival law, random assignment to queues
      CrossQueue (NONE); // each car stays in its queue
      Statistics (WT, OR, MQL); // wait time, occupancy rate, maximum queue length
      WrapUp (SMOOTH); // when the time is up, stop taking new cars, but serve cars in line
     }
```

Ideally, one may want to define a formal Application Modeling Language, and build a compiler for it, in such a way that an application description such as this could be compiled into a finished application.

3.2. A Reference Architecture

The foregoing discussion yields a natural reference architecture for the proposed product line, whose main components include instances of the queue/station structure (QS), an Arrivals component, a Statistics component, and a main program to coordinate all these; this is illustrated in Fig. 1, below. This figure describes the dataflow between these components; as for the control flow, it is basically limited to the main component invoking all the others in a sequential manner.

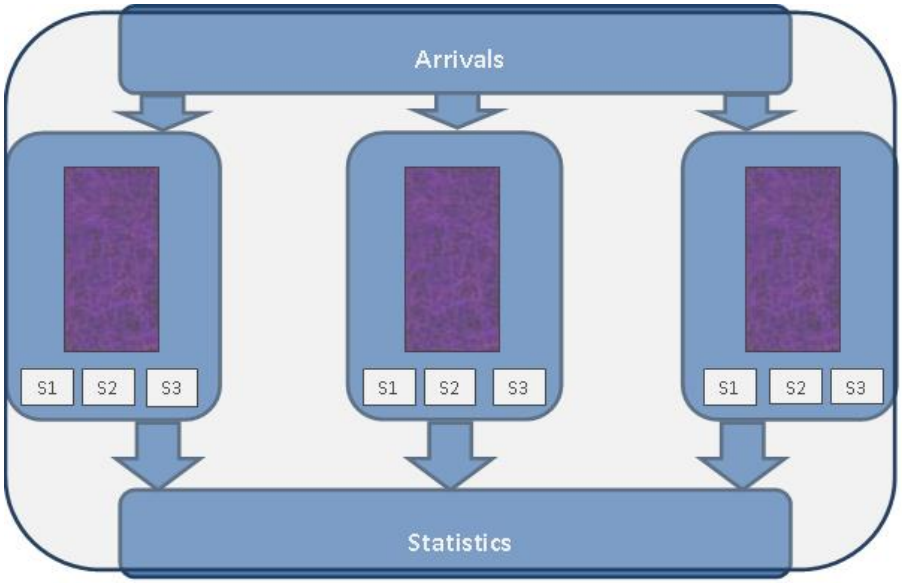

Fig. 1. Architecture.

We review the variabilities that we have listed in Section 2 and see how these map onto this architecture; in other words, once we decide on the value of a variability, which component must be modified and how. We consider the variabilities, in turn:

- *Topology of service stations*. This variability affects the main program, as well as the queue/station instances. This variability determines the number of QS instances we create, and the number of service stations we declare for each.
- *Service Time*. This variability affects the Arrivals component, as it determines how service requests are

computed for incoming customers.

- *Topology of queues*. This variability affects the main program, as well as the queue/station instances. This variability determines the number of QS instances we create, and how queues are associated with service stations.
- *Arrival Distribution*. This variability affects the Arrivals component.
- *Queuing Policy*. This variability affects the declared instances of the QS components, as well as the relationships between them.
- *Dispatching Policy*. This variability affects the declared instances of the QS components, in the sense that it determines how idle stations determine where their next customer comes from; it also affects whether an idle station may take a customer from another QS component.
- *Measurements*. This variability affects the Statistics component, by determining what functions to collect data for during the simulation and to summarize at the end of the simulation.
- *Wrap Up Policy*. This variability affects the main program, in the sense that it determines the main simulation loop of the main program, by dictating the exit condition of the loop.

3.3. Domain Implementation

We choose to implement this product line in C++. In this section, we outline the broad structure of the main program, and then we implement the main building blocks that are used in this product line. The main program reads as follows:

```
#include <iostream>                                                    line 1
#include "qs.cpp"                                                        2
#include "arrivals.cpp"                                                  3
#include "statistics.cpp"                                                4
using namespace std;                                                     5
typedef int clocktimetype;                                               6
typedef int durationtype;                                                7
      /*   State Objects */                                              8
      qsclass qs1, qs2;                                                  9
      arclass arrivals;                                                 10
      stclass stats;                                                    11
      /*   State Variables      */                                      12
      durationtype expduration;                                         13
      int arrivalrate1, servicerate1;   // for class 1                  14
      int arrivalrate2, servicerate2;   // for class 2                  15
      int nbcustomers;                                                   16
      /*   Working Variables   */                                       17
      clocktimetype clocktime;                                          18
      customertype customer;                                            19
      bool newarrival;                                                  20
      int locsum, locmin, locmax;                                       21
/*   functions   */                                                     22
bool ongoingSimulation();                                               23
void elicitParameters();                                                24
int main ()                                                             25
    {                                                                   26
      elicitParameters();                                               27
      while (ongoingSimulation())                                       28
          {                                                             29
            arrivals.drawcustomer(clocktime, expduration,               30
                arrivalrate1, servicerate1, customer, newarrival);      31
            if (newarrival) {nbcustomers++; qs1.enqueue(customer);}     32
            arrivals.drawcustomer(clocktime, expduration,               33
```

```
            arrivalrate2, servicerate2, customer, newarrival);                    34
          if (newarrival) {nbcustomers++; qs2.enqueue(customer);}                   35
          qs1.update(clocktime, locsum, locmin, locmax);                            36
          stats.record(locsum,locmin,locmax);                                       37
          qs2.update(clocktime, locsum, locmin, locmax);                            38
          stats.record(locsum,locmin,locmax);                                       39
          clocktime++;                                                              40
        };                                                                          41
    cout << "concluded at time: " << clocktime << endl;                          42
    stats.summary(nbcustomers);                                                     43
  }                                                                                 44
bool ongoingSimulation()                                                            45
  {                                                                                 46
    return ((clocktime<=expduration) ||                                             47
            (!qs1.done()) ||   (!qs2.done()));                                      48
  };                                                                                49
void elicitParameters()                                                             50
  {                                                                                 51
    nbcustomers=0;                                                                  52
    cout << "Length of Simulation" << endl;                                         53
    cin >> expduration;                                                             54
    cout << "Arrival rate, Service Rate, Station 1" << endl;                    55
    cin >> arrivalrate1 >> servicerate1;                                        56
    cout << "Arrival rate, Service Rate, Station 2" << endl;                    57
    cin >> arrivalrate2 >> servicerate2;                                        58
  };                                                                                59
```

As written, this main program refers to two identical QS components; but in general, it may refer to more than one type of QS components (as we recall, QS components may differ by their number of stations and their queuing policy). Also, this main program refers to an arrivals component, that determines the rate of customer arrivals, and a statistics component, that collects statistics. For the sake of simplicity, we have opted for a straightforward (tree-like) #include hierarchy between the various components of this program; the price of this choice is that most data has to transfer through the main program rather than directly between the subordinate components. Hence the decision of whether there is a new arrival and the selection of the new customer parameters transits through the main program (lines 30-31 and 32-34) on its way to the QS component that stores incoming customers (lines 32 and 35); likewise, statistical data is sent from the QS components (lines 36 and 38) to the statistics components (lines 37 and 39) via the main program. Note that whereas the topology and configuration of the queues and service stations is decided at application engineering time, the actual simulation parameters (experiment duration, arrival rate of each class of customers, service rate of each class of customers) are decided at run-time (line 27).

The QS component is defined by the following header file:

```
//*********************************************************
//   Header file qs.h
//
//*********************************************************
const int maxq = 1000;         // max size of queue                    line 1
const int nbs    = 3;          // number of stations for single queue    2
const int largewait=2000;    // used for min wait                        3
typedef int clocktimetype;                                               4
typedef int servicetimetype;                                             5
typedef int durationtype;                                                6
typedef int customeridtype;                                              7
typedef int indextype;                                                   8
```

```
typedef struct                                                    9
    {customeridtype cid;                                         10
     clocktimetype at;                                           11
     servicetimetype st;                                         12
     int ccat;                                                   13
    }   customertype;                                            14
typedef struct                                                   15
    {customertype guest;                                         16
     durationtype busytime;                                      17
     int busyrate;                                               18
    } stationtype;                                               19
class qsclass                                                    20
    {public:                                                     21
        qsclass ();     // default constructor                   22
        bool done ();                                            23
        void update (clocktimetype clocktime,                    24
                     int& locsum, int& locmin, int& locmax);     25
        bool emptyq () const;   // tells whether q is empty     26
        void enqueue (customertype qitem);                       27
        void dequeue ();                                         28
        void checkfront (customertype& qitem) const;             29
        int queuelength ();                                      30
                                                                 31
     private:                                                    32
        customertype qarray [maxq];                              33
        stationtype   sarray [nbs];                              34
        indextype front;                                         35
        indextype back;                                          36
        int qsize;                                               37
    };                                                           38
```

The nbs parameter in this header file (line 2) indicates the number of service stations in the QS component; ideally, we would like to define a single QS component for each queuing policy (e.g. FIFO), and let nbs be a parameter (hence for example writing QS(3) or QS(5) depending on the number of stations we want to have for each queue) but we do not believe C++ allows that; hence in practice we write a separate QS class for each different value of nbs and each different value of the queuing policy; in the case of this product line, if we adopt FIFO as the only queuing policy and nbs=3 as the only viable number of stations per queue, then only one QS class is needed. The state variables of this class include the queue infrastructure (qarray, front, back, qsize) as well as an array of service stations, of size nbs. In addition to the queue methods (emptyq, queuelength, checkfront, enqueue, dequeue), this class has two QS-specific methods, which are (Boolean-valued) done() and (void) update(clocktime, locsum, locmin, locmax). The former indicates that the QS component has no residual customers in its queue or its service stations; the latter updates the queue and service stations on the grounds that a new unit of time (minute, second, millisecond, etc) has elapsed:

- If a service station is still busy, it updates the remaining busy time thereof.
- If a service station has just completed serving a customer, it frees the customer and collects statistical data on it.
- If a service station is free and the queue is not empty, then it dequeues the customer at the front of the queue and loads it on the service station.

The arrivals component reads as follows:

```
************************************************************      line 1
//                                                                  2
```

```
//    arrivals component;                                                    3
//    file arrivals.cpp, refers to header file arrivals.h.                   4
//                                                                           5
// ********************************************************                  6
#include "arrivals.h"                                                       7
#include "rand.cpp"                                                         8
arclass :: arclass ()                                                       9
    {SetSeed(673); customerid=1001;                                         10
    };                                                                      11
void arclass :: drawcustomer (clocktimetype clocktime, int expduration,     12
                              int arrivalrate, int servicerate,             13
                              customertype& customer, bool& newarrival)     14
    {float draw = NextRand();                                               15
     newarrival = ((clocktime<=expduration) &&                              16
                   (draw<(1.0/float(arrivalrate))));                        17
     if (newarrival)                                                        18
        {customer.cid = customerid; customerid = customerid+3;              19
         customer.at = clocktime;                                           20
         customer.st = 1+int(NextRand()*servicerate);                       21
        }                                                                   22
    }                                                                       23
```

This component calls a random number generator using the parameters of arrival time to determine whether or not there is an arrival at time clocktime, and if there is an arrival, it uses the parameter of service time to draw the length of service needed by the new arriving customer, assigns it a customer ID, and timestamps its arrival time. This information is used subsequently for reporting purposes and/ or to compute statistics.

The header of the statistics component reads as follows:

```
//*********************************************************          line 1
//    Header file statistics.h                                             2
//                                                                         3
//*********************************************************              4
class stclass                                                            5
    {public:                                                             6
        stclass ();      // default constructor                          7
        void summary (int nbcustomers);                                  8
        void record (int locsum, int locmin, int locmax);                9
     private:                                                            10
        int totalwait, minwait, maxwait;                                 11
        int totalstay, minstay, maxstay;                                 12
    };                                                                   13
```

As written, this component maintains some information about wait times and stay times of customers in the system; it is adequate if all we are interested in are statistics about these two quantities; but it needs to be expanded if we are to support all the variabilities listed in Section 3.2. As written, this component has two main functions:

- Collecting data pertaining to wait times and stay times, which transits through the main program (rather than directly from the QS components).
- Summarizing the collected data and printing it to the output at the end of the simulation.

4. Testing at Domain Engineering

The question that we address in this section is: now that we have implemented our sample product line, how do we test it? The issues we raised in the introduction about testing product lines at domain

engineering are that

- First, we have no executable code to test.
- Second we have no specific requirements to test our code against, since we have developed a generic product line, rather than a fully specified individual application.

In regards to the first issue, we agree that because of the provisions we made for variabilities, not all design decisions are finalized; but we argue that we can test selected versions of the code, to gain some confidence in its correctness and/or to find and remove faults therein. As for the second issue, we argue that whereas we cannot pin down a specification for the whole product line, we can do so for individual components; hence we propose to take individual components, build dedicated test drivers for them, and test them to an arbitrary level of thoroughness, as a way to gain confidence in the correctness and soundness of the product line.

As an illustration, we consider for example the arrivals components, and write a test driver for it, in such a way that its behavior can be checked easily. For example, if we invoke the arrivals component 10000 times with an arrival rate of 4 and a service rate of 20, then we expect to generate about 2500 new customers whose average service rate is about 10 units of time. Note that to write this test driver, we need not see the file arrivals.cpp (in fact it is better not to, for the sake of redundancy); we only need to see the (specification) file arrivals.h. We propose the following test driver:

```
#include <iostream>                                                    line 1
#include "qs.cpp"                                                        2
#include "arrivals.cpp"                                                  3
using namespace std;                                                     4
typedef int clocktimetype;                                               5
typedef int durationtype;                                                6
        /*    State Objects */                                           7
        arclass arrivals;                                                8
       /*    Working Variables    */                                     9
       customertype customer; bool newarrival;                          10
       int nbcustomers;    durationtype totalst;                        11
int main ()                                                             12
    {for (int clocktime=1; clocktime<=10000; clocktime++)               13
          {arrivals.drawcustomer(clocktime,10000,4,20,customer,newarrival);    14
            if (newarrival) {nbcustomers++; totalst=totalst+customer.st;}      15
          };                                                                   16
      cout << "nb customers: " << nbcustomers << endl;                  17
      cout << "average service time: " << float(totalst)/nbcustomers << endl;18
    }                                                                        19
```

Execution of this test driver yields the following output,

- nb customers: 2506
- Average service time: 10.7027

which corresponds to our expectation, and enhances our faith in the arrivals component.

We could, likewise, test the QS component by generating and storing a number of customers in its queue, then monitoring how it handles the load. For example, we can generate 300 customers that each requires 20 units of service time, and see to it that it schedules them in 2000 minutes. We consider the following test driver:

```
#include <iostream>                                                    line 1
#include "qs.cpp"                                                        2
#include "statistics.cpp"                                                3
using namespace std;                                                     4
```

```
                                                                    5
typedef int clocktimetype;                                          6
typedef int durationtype;                                           7
                                                                    8
/*    State Objects */                                              9
qsclass qs;                                                         10
stclass stats;                                                      11
/*    Working Variables    */                                       12
clocktimetype clocktime;                                            13
customertype customer;                                              14
durationtype expduration; int nbcustomers;                          15
int locsum, locmin, locmax;                                         16
/*    functions    */                                               17
bool ongoingSimulation();                                           18
                                                                    19
int main ()                                                         20
    {clocktime=0;                                                   21
     expduration = 0; // terminate whenever qs is empty             22
     customer.cid=1001; customer.at=0; customer.st=20;              23
     for (int i=1; i<=300; i++) {qs.enqueue(customer);};            24
     nbcustomers=300;                                               25
     while (ongoingSimulation())                                    26
         {qs.update(clocktime, locsum, locmin, locmax);             27
          stats.record(locsum,locmin,locmax);                       28
          clocktime++;                                              29
         };                                                         30
     cout << "concluded at time: " << clocktime << endl;            31
     stats.summary(nbcustomers);                                    32
    }                                                               33
bool ongoingSimulation()                                            34
    {return ((clocktime<=expduration) ||(!qs.done()));              35
    };                                                              36
```

The outcome of the execution of this test driver is the following output, which corresponds to our expectation: The 300 customers kept the 3 stations busy non-stop for a total of 100×20 minutes, i.e. 2000 minutes. The sum of waiting times is an arithmetic series, of the form

$$(1 + 2 + 3 + \cdots + 99)\times 20.$$

Dividing this sum by the number of customers on each station (100) and replacing the arithmetic series by its closed form expression, we find

$$\frac{99\times 100\times 20}{2\times 100} = 990$$

The minimum waiting time is 0, of course since the stations were available at the start of the experiment. The maximum waiting time is the waiting time of the last customer in each queue, which had to wait for the 99 customers before it, hence the maximum waiting time is 99×20=1980. All this is confirmed in the following output, delivered by the test driver (except for the minor detail that the test driver shows the closing time at 2001 rather than 2000, but that is because the main loop increments the clocktime at the bottom of the loop body).

- Concluded at time: 2001
- nb customers 300
- Statistics, wait time (total, avg, min, max):

- 297000 990 0 1980.

Interestingly, running this test driver enabled us to uncover and remove a fault in the code of the QS component, which was measuring stay time rather than wait time. As far as domain engineering is concerned, we can perform testing under the following conditions:

- We test individual components rather than whole applications. Individual components lend themselves more easily to simple, compact specifications, which can be used as oracles.
- We test preferably components that have the least variability, or whose variabilities are trivial. Ideally, we want any confidence we gain about the correctness of a component to survive as the component is adapted to other applications.

In our case study, we have tested component qs with nbs=3; we can be reasonably confident that changing the value of nbs for the purposes of another application does not alter significantly the confidence we have in its correctness. But changing the queuing policy, however, (e.g. from FIFO to priority) will require a new testing effort. We are able to single out individual components, write test drivers for them, and design targeted test data that will exercise specific functionalities, and for which we know exactly what outcome to expect.

All the testing effort that we carry out at the domain engineering phase is expended once, but will benefit every application that is produced from this product line.

5. Testing at Application Engineering

In the application engineering phase, we take the domain engineering assets and use them to build an application on the basis of specific requirement specifications. In application engineering, we avail ourselves of an executable product for which we have a product specification; hence we have everything we need to run a test; the issue here is to maximize return on investment by targeting test data to those aspects of the application that have not been adequately tested at domain engineering or have not been adequately covered by the test of other applications within the same domain.

We consider a sample application in our queue simulation domain, specified by the following AML statements:

```
Simulation airlineCheckin
    {
       QS(FIFO, 4)    coach;
       QS(FIFO, 2)    firstClass;
       CrossStation(NONE);              // each class services its own queue
       Arrivals (UNIFORM, CAT);         // arrival law, assignment by category
       CrossQueue(NONE);                // each passenger stays in his/her queue
       Statistics (WT);                 // wait time
       WrapUp (SMOOTH);                 // when check-in ends, take no new passengers, but clear lines
          }
```

In light of this specification, we propose to define two QS classes, one with four service stations, and one with two service stations. Also, we envision that the user specifies the arrival rate and service rate of each class of service (coach, first class), and to deliver statistics about wait times. This yields the following simulation program.

```
#include <iostream>
#include "qs2.cpp"
#include "qs4.cpp"
#include "arrivals.cpp"
#include "statistics.cpp"
```

```
using namespace std;

typedef int clocktimetype;
typedef int durationtype;
/*    State Objects */
qsclass2 qs2;
qsclass4 qs4;
arclass arrivals;
stclass stats;
/*    State Variables       */
durationtype expduration;
int arrivalrate2, servicerate2;    // for class 2
int arrivalrate4, servicerate4;    // for class 4
int nbcustomers;
/*    Working Variables    */
clocktimetype clocktime;
customertype customer;
bool newarrival, newdeparture;
int locsum, locmin, locmax;
/*    functions    */
bool ongoingSimulation();
void elicitParameters();
int main ()
    {elicitParameters();
     while (ongoingSimulation())
          {arrivals.drawcustomer(clocktime, expduration,
                arrivalrate2, servicerate2, customer, newarrival);
           if (newarrival) {nbcustomers++; qs2.enqueue(customer);}
           arrivals.drawcustomer(clocktime, expduration,
             arrivalrate4, servicerate4, customer, newarrival);
           if (newarrival) {nbcustomers++; qs4.enqueue(customer);}
         qs2.update (clocktime, locsum, locmin, locmax);
         stats.record (locsum,locmin,locmax);
         qs4.update (clocktime, locsum, locmin, locmax);
         stats.record(locsum,locmin,locmax);
            clocktime++;
          };
     cout << "concluded at time: " << clocktime << endl;
     stats.summary(nbcustomers);
    }
bool ongoingSimulation()
    {return ((clocktime<=expduration)||(!qs2.done())||(!qs4.done()));
    };

void elicitParameters()
    {nbcustomers=0;
     cout << "Length of Simulation" << endl;
     cin >> expduration;
     cout << "Arrival rate, Service Rate, First Class" << endl;
     cin >> arrivalrate2 >> servicerate2;
     cout << "Arrival rate, Service Rate, Coach" << endl;
     cin >> arrivalrate4 >> servicerate4;
    };
```

Successive executions of this program with different arrival rates and service rates give the following results, illustrated in Table 1.

Due to the random nature of customer generation, we have no way to check the details of the output for each set of input data, but it is clear from Table 1 that the program behaves in a credible manner according

to the simulation parameters.

Table 1. Results

Duration	First Class		Coach		Time Ended	Customers Served	Average wait	Max wait
	Arr Rate	Ser Rate	Arr Rate	Ser Rate				
900	4	24	3	19	1573	542	152	665
900	4	20	3	15	1321	542	95	415
900	4	16	3	12	1076	542	41	173
900	4	14	2	7	987	688	13	92
900	4	10	2	5	906	688	0.89	13
1000	5	20	4	14	1086	465	21	82

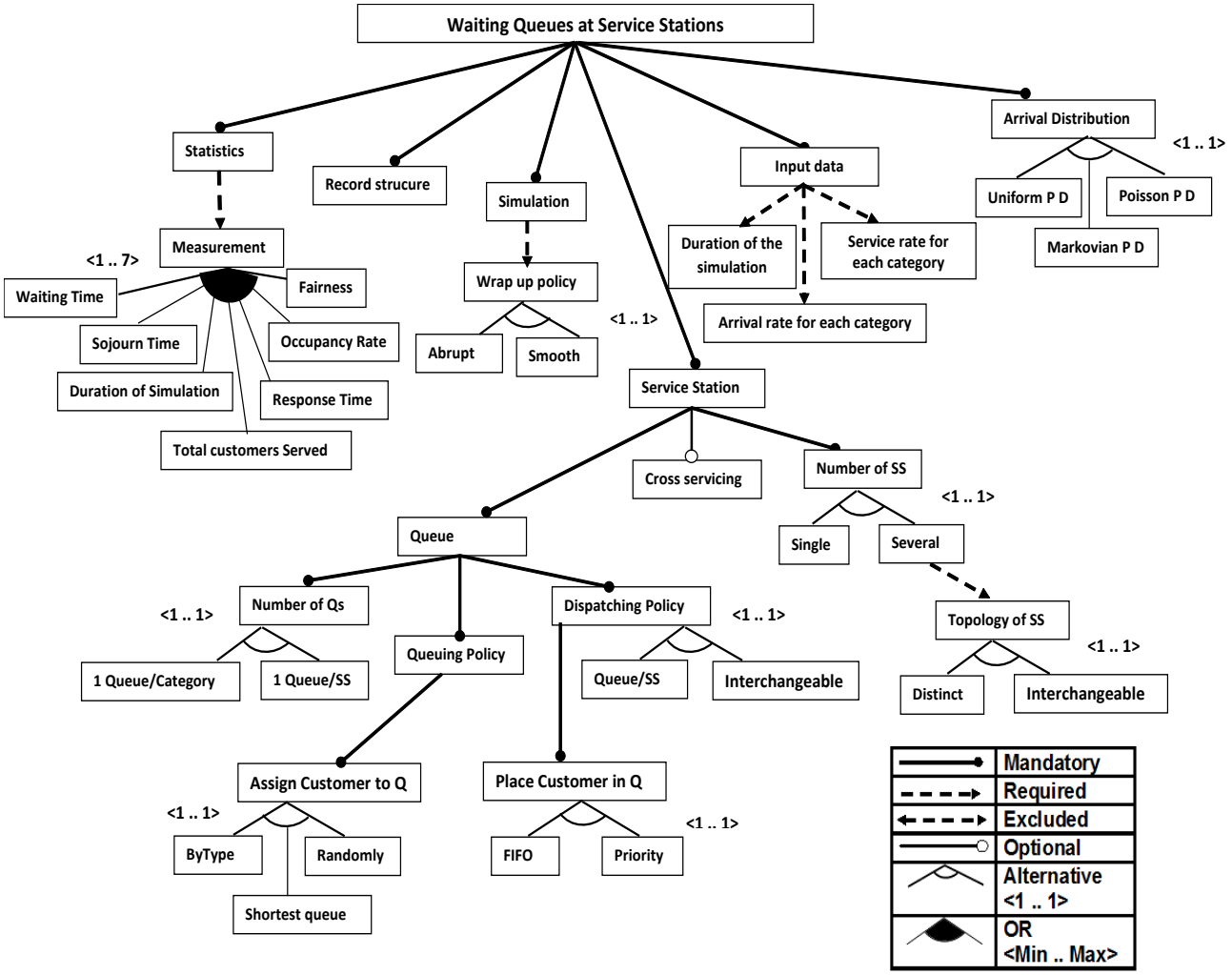

Fig. 2. Feature model.

6. Methods and Strategies of SPL Testing

In this section we discuss two methods for generating test cases from use cases [4] or PLUCs (Product Line Use Cases) [4], [5] (see in Fig. 3). Also, we discuss three strategies for SPL testing [4].

6.1. Methods of SPL Testing

6.1.1. Heumann's method

Generate test cases from use cases as follows [4]:

- Generation of scenarios which contain all the possible combinations between the normal flow and the alternative flows of the application described in the use cases.
- Identification of the test data from the list of scenarios and use cases.
- Identification of the test values that consists on replacing the valid and invalid values by actual values respectively, obtaining the matrix of test cases.

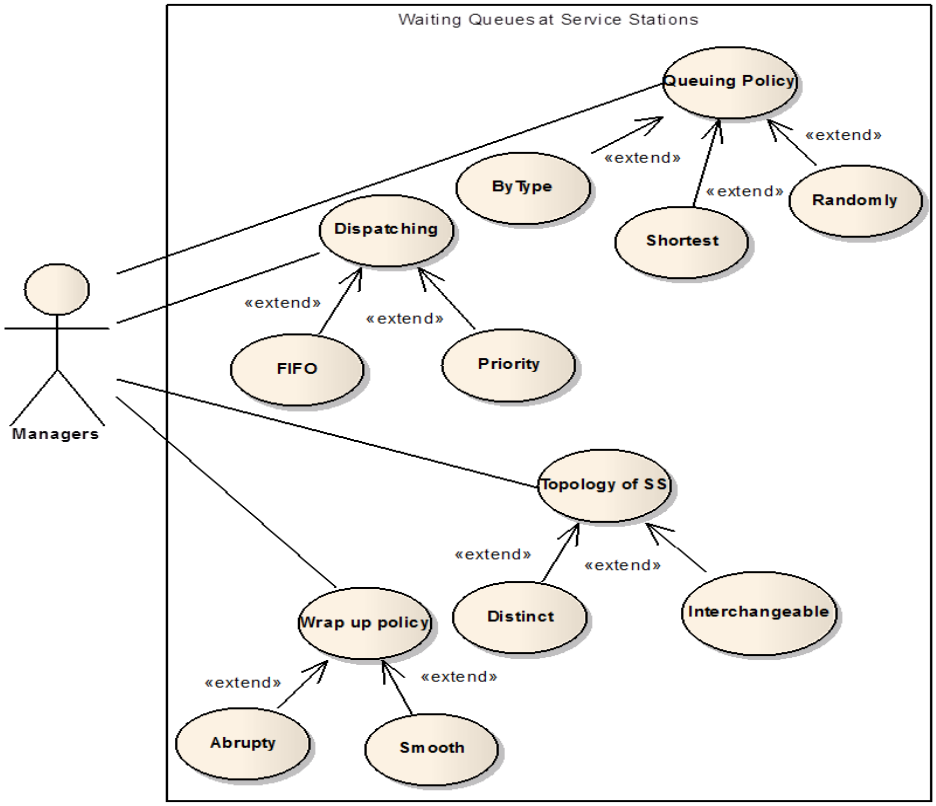

Fig. 3. Some use cases for waiting queues at service stations.

PL USE CASE QueuingPolicy	
Goal:	Assign the arriving customers to the queue [V0].
Scope:	The [V0] queue simulation.
Level:	Summary.
Precondition:	newarrival equal true
Trigger:	Generate new customer.
Primary actor:	The main component (appeng class).
Main success scenario:	1. there is a new arrival customer. 2. the main component assign arrived customer to [V1] and increment the clocktime.
Extensions:	1.a. there is no a new arrival customer 1.a.1the main component increment the clocktime.
Variations	
V0: Alternative	
	0: By Type 1: Shortest 2: Randomly
V1: Parametric	
If V0=0 then a particular type of queue else if V0=1 then shortest queue else if V0=2 then queue randomly.	

Fig. 4. PLUC for queuing policy.

6.1.2. Product line use case test optimisation (PLUTO)

Generate test cases from use cases called PLUCs (Product Line Use Cases) as follows [4], [5]:

- Generation of scenarios for each use case (main scenario and extension scenarios) (see in Fig. 4-Fig. 7).
- Generation of categories: selected considering relevant information from the tester's viewpoint (see in Fig. 8-Fig. 11).
- Generation of choices: values that a category can take (see in Fig. 12, Fig. 13, and Fig. 14).

PL USE CASE DispatchingPolicy	
Goal:	Pick customers for service whenever a service station becomes available using [V0] policy.
Scope:	The [V0] queue simulation.
Level:	Summary.
Precondition:	Service station busy time equal zero and the queue is not empty.
Trigger:	Service station becomes available and there are customers waiting for service.
Primary actor:	The main component (appeng class).
Main success scenario:	1. If a service station is free and the queue is not empty, 2. QS component dequeues [V1] and loads it on the service station. 3. then shift the remains customers in the queue to the front.
Extensions:	1.a. If a service station is still busy, 1.a.1 QS component updates the remaining busy time thereof. 1.b. If a service station has just completed serving a customer, 1.b.1 QS component frees the customer and collects statistical data on it.
Variations	
V0: Alternative	
	0: FIFO
	1: Priority
V1: Parametric	
if V0=0 then the customer at the front of the queue else if V0=1 then the customer with highest priority.	

Fig. 5. PLUC for dispatching policy.

PL USE CASE TopologyOfSS	
Goal:	Configure the service stations to be [V0].
Scope:	The [V0] queue simulation.
Level:	Summary.
Precondition:	Service station busy time equal zero and the queue is not empty.
Trigger:	Service station becomes available and there are customers waiting for service.
Primary actor:	The main component (appeng class).
Main success scenario:	1. If a service station is free and the queue is not empty, 2. the QS component select the [V1] in the queue to served. 3. then shift the remains customers in the queue to the front.
Extensions:	1.a. If a service station is still busy, 1.a.1 QS component updates the remaining busy time thereof. 1.b. If a service station has just completed serving a customer, 1.b.1 QS component frees the customer and collects statistical data on it.
Variations	
V0: Alternative	
	0: Interchangeable
	1: Distinct
V1: Parametric	
if V0=0 then first customer else if V0=1 then customer has the same type of service station.	

Fig. 6. PLUC for topology of SS.

PL USE CASE WrapUpPolicy

Goal:	End the simulation [V0].
Scope:	The [V0] queue simulation.
Level:	Summary.
Precondition:	clocktime greater than or equal expduration {[V2] and queue is empty}
Trigger:	The expected duration is elapsed {[V2] and there are no residual customers in the queues }
Primary actor:	The main component (appeng class).
Main success scenario:	1. in each iteration the main component check [V1] to exit from the loop. 2. in case the result true, stop the simulation {[V2] no new customers are generated and the simulation continues until all the current customers have been serviced and have exited the system}.
Extensions:	2.a. in case the result false, continue in the iteration.

Variations

V0: Alternative

0: Abruptly
1: Smoothly

V1: Parametric

If V0=0 then the selected time elapses else if V0=1 then the time of the simulation is exhausted and there are no residual customers in the queues.

V2: Optional

when V0=1

Fig. 7. PLUC for Wrap up policy.

PLUC DispatchingPolicy Test Specification

[V0]: Dispatching Policy:

0: FIFO [Property P0]
1: Priority

Queue:
qsize
front
maxq [if P0]
back [if NOT P0]

Customer:
Priority [if NOT P0]

Scienarios:
Main
Ext: S S still busy
Ext: S S completed serving a customer

Fig. 8. Test categories for the DispatchingPolicy PLUC.

PLUC TopologyOfSS Test Specification

[V0]: Topology of Service Station:

1: Interchangeable [Property P0]
2: Distinct

Queue:
qsize
front
back [if NOT P0]
maxq [if P0]

Service Station:
station_type [if NOT P0]

Customer:
customer_type [if NOT P0]

Scienarios:
Main
Ext: S S still busy
Ext: S S completed serving a customer

Fig. 9. Test categories for the TopologyOfSS PLUC.

PLUC QueuingPolicy Test Specification

[V0]: Queuing Policy:

0: ByType [Property P0]
1: Shortest
2: Randomly [Property P2]

Queue:
queue_type [if NOT P2][if P0]
queue_back [if NOT P0] [if NOT P2]
queue_random [if NOT P0][if P2]
maxq [if NOT P0][if P2]

Customer:
clocktime
nbcustomers

Arrival:
newarrival

Scienarios:
Main
Ext

Fig. 10. Test categories for the QueuingPolicy PLUC.

PLUC WrapUpPolicy Test Specification

[V0]: Warp Policy:

0: Abruptly [Property P0]
1: Smoothly

Arrival:
expduration

Customer:
clocktime

Service station:
busytime [if NOT P0]

Scienarios:
Main
Ext
[V2]:Queues:
queue is empty [if NOT P0]
queue is not empty [if NOT P0]

Fig. 11. Test categories for the WrapUpPolicy PLUC.

Product 1:	
1. All commonalities:	
2. Variablity1:	
QueueingPolicy:	
1. The queue for arriving customer selected:	ByType
2. Queue:	Assigned to a particular type of queue.
3. Scenario:	Main
DispatchingPolicy	
1. Policy:	FIFO
2. The place in the selected queue:	The end of the selected queue, after last customer.
3. Scenario:	Main
TopologyOfSS	
1. Configuration of service station	Distinct
2. Services:	Different services.
3. Scenario:	Main
WrapUpPolicy	
1. The simulation winds down	Abrupt
2. Simulation time:	The time elapses, flushed out all remaining customers.
3. Simulation:	Simulation off.
4. Scenario:	Main

Fig. 12. Some scenarios for Product 1.

Product 2:	
1. All commonalities:	
2. Variablity1:	
QueueingPolicy:	
1. The queue for arriving customer selected:	Randomly
2. Queue:	Randomly assigned to a queue that corresponds to his category.
3. Scenario:	Main
DispatchingPolicy	
1. Policy:	FIFO
2. The place in the selected queue:	The end of the selected queue, after last customer.
3. Scenario:	Main
TopologyOfSS	
1. Configuration of service station	Interchangeable
2. Services:	Same services.
3. Scenario:	Main
WrapUpPolicy	
1. The simulation winds down	Abrupt
2. Simulation time:	The time elapses, flushed out all remaining customers.
3. Simulation:	Simulation off.
4. Scenario:	Main

Fig. 13. Some scenarios for Product 2.

Product 3:	
1. All commonalities:	
2. Variablity1:	
QueueingPolicy:	
1. The queue for arriving customer selected:	Randomly
2. Queue:	Randomly assigned to a queue that corresponds to his category.
3. Scenario:	Main
DispatchingPolicy	
1. Policy:	Priority
2. The place in the selected queue:	The end of the selected queue, after last customer.
3. Scenario:	Main
TopologyOfSS	
1. Configuration of service station	Distinct
2. Services:	Different services.
3. Scenario:	Main
WrapUpPolicy	
1. The simulation winds down	Smooth
2. Simulation time:	The time is exhausted, no new customers are generated
3. Simulation:	Simulation on until all current customers serviced and exit. Then the simulation off.
4. Scenario:	Main

Fig. 14. Some scenarios for Product 3.

6.2. Strategies of SPL Testing

6.2.1. Product by product

The testing for each product individually, without reuse the test cases that are developed for the previous products [4].

In the queue simulation example, we have used the Feature Model as standard [6] to illustrate the variability (see in Fig. 2): firstly test product 1, derive all the test cases for all common and variable features. Then test product 2, also derive all the test cases for all common and variable features and so on (shown in Fig. 15).

The test case generation starts in Application Engineering as follow:

- Product 1 (shown in Fig. 12):

1) Generate test cases for all common PLUCs ✓
2) Generate test cases for QueuingPolicy PLUC ✓
3) Generate test cases for DispatchingPolicy PLUC ✓
4) Generate test cases for TopologyOfSS PLUC ✓
5) Generate test cases for WrapUpPolicy PLUC ✓

- Product 2 (shown in Fig. 13):

1) Generate test cases for all common PLUCs ✓
2) Generate test cases for QueuingPolicy PLUC ✓
3) Generate test cases for DispatchingPolicy PLUC ✓
4) Generate test cases for TopologyOfSS PLUC ✓
5) Generate test cases for WrapUpPolicy PLUC ✓

- Product 3 (shown in Fig. 14):

1) Generate test cases for all common PLUCs ✓
2) Generate test cases for QueuingPolicy PLUC ✓
3) Generate test cases for DispatchingPolicy PLUC ✓
4) Generate test cases for TopologyOfSS PLUC ✓
5) Generate test cases for WrapUpPolicy PLUC ✓

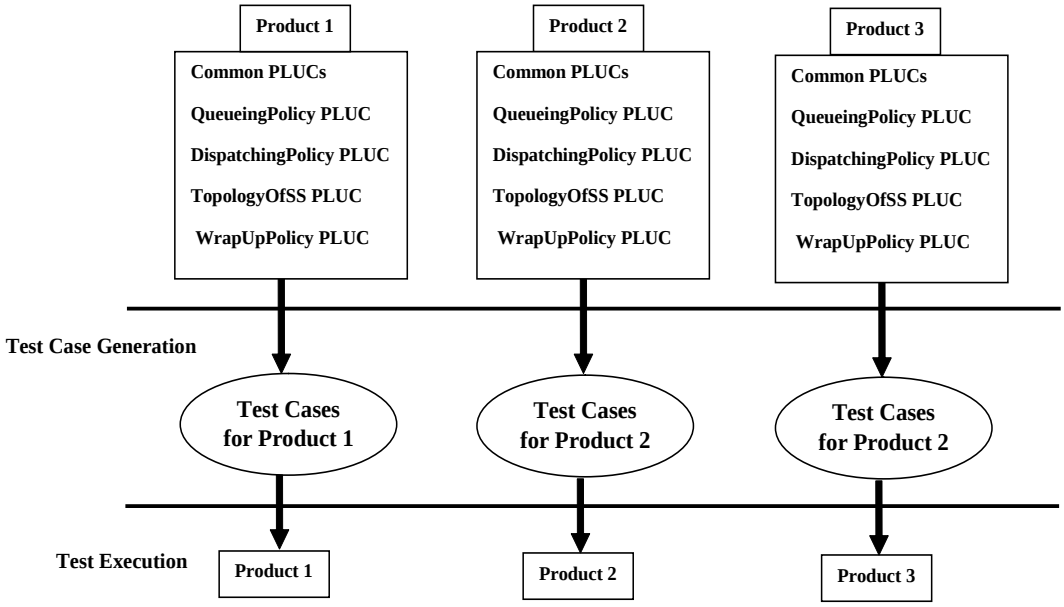

Fig. 15. Product by product.

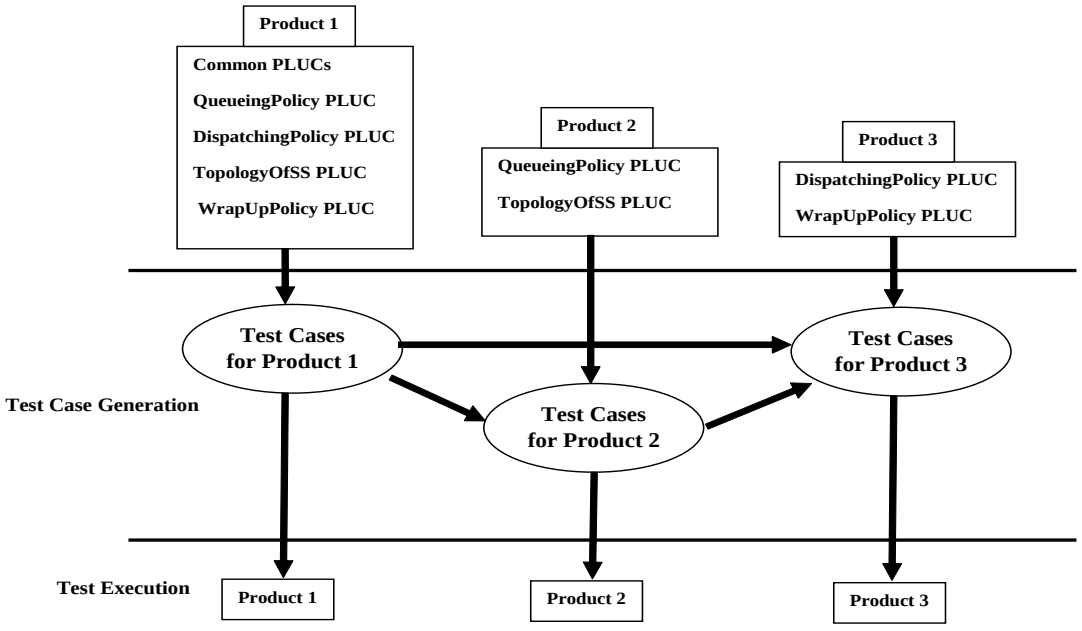

Fig. 16. Incremental strategy.

6.2.2. Incremental strategy

The testing started with the first product, then the test cases are reused for the next product with developing some test cases for the variable parts that is related to the next product [4].

In the queue simulation example, we have used the Feature Model as standard [6] to illustrate the variability (shown in Fig. 2): firstly test product 1, derive all the test cases related to it. Then for product 2 reuse all the common features between product 1 and product 2 and develop test cases for the variable features (features in product 2 not in product 1) and so on (see in Fig. 16).

The test case generation starts in Application Engineering as follow:

- Product 1 (shown in Fig. 12):

1) Generate test cases for all common PLUCs ✓
2) Generate test cases for QueuingPolicy PLUC ✓
3) Generate test cases for DispatchingPolicy PLUC ✓
4) Generate test cases for TopologyOfSS PLUC ✓
5) Generate test cases for WrapUpPolicy PLUC ✓

- Product 2 (shown in Fig. 13):

1) Reuse test cases for all common PLUCs from Product 1
2) Reuse test cases for DispatchingPolicy PLUC from Product 1
3) Reuse test cases for WrapUpPolicy PLUC from Product 1
4) Generate test cases for QueuingPolicy PLUC ✓
5) Generate test cases for TopologyOfSS PLUC ✓

- Product 3 (shown in Fig. 14):

1) Reuse test cases for all common PLUCs from Product 1
2) Reuse test cases for QueuingPolicy PLUC from Product 2
3) Reuse test cases for TopologyOfSS PLUC from Product 1
4) Generate test cases for DispatchingPolicy PLUC ✓
5) Generate test cases for WrapUpPolicy PLUC ✓

6.2.3. Reusable asset instantiation

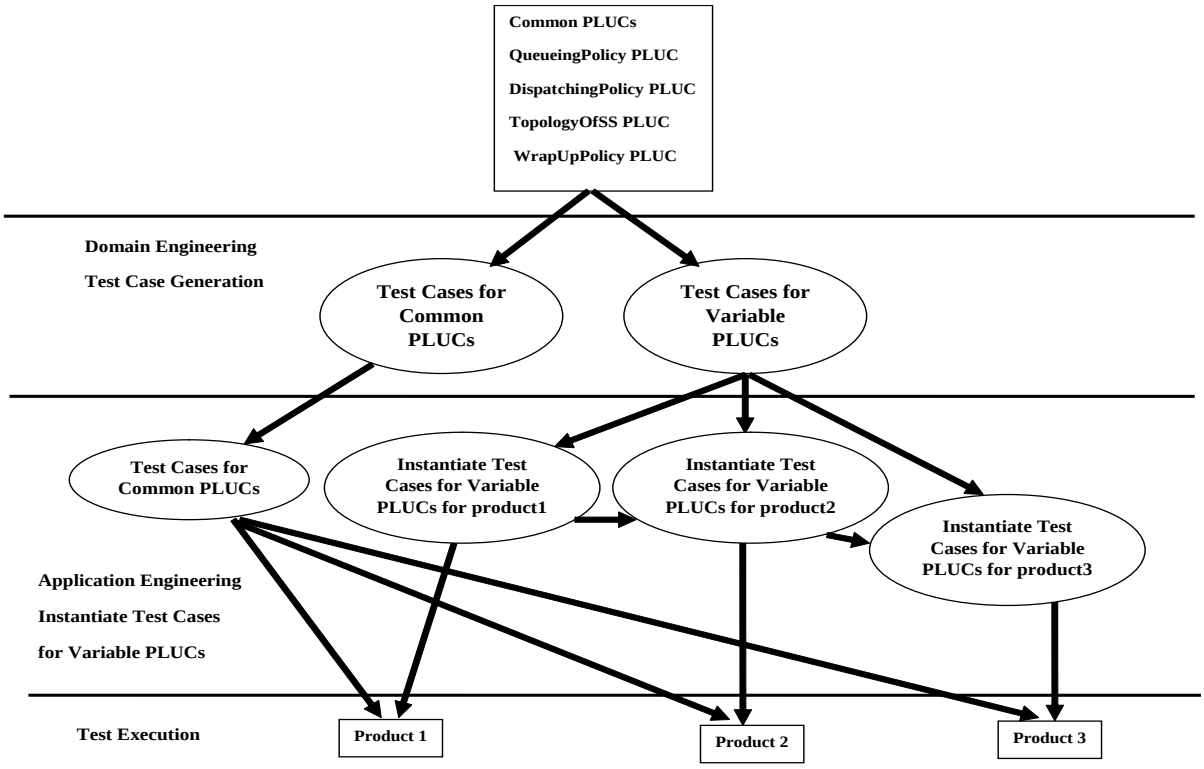

Fig. 17. Reusable asset instantiation.

The testing started in the domain engineering phase for all the features (common and variable parts) (shown in Fig. 2). Then for each product in SPL, reuse the test cases for common parts and instantiate the test cases for the variable parts that are generated in the domain engineering [4].

In the queue simulation example, we have used the Feature Model as standard [6] to illustrate the variability (see in Fig. 2): the test cases for all features (common and variable parts) are generated from the domain engineering phase. Then product 1, reuse all the test cases for common features and instantiate the test cases for variable features (features specific for product 1) and so on (see in Fig. 17).

The test case generation starts in Domain Engineering as follow:

- Generate test cases for all common PLUCs ✓
- Generate test cases for QueuingPolicy PLUC ✓
- Generate test cases for DispatchingPolicy PLUC ✓
- Generate test cases for TopologyOfSS PLUC ✓
- Generate test cases for WrapUpPolicy PLUC ✓

In Application Engineering:

- Product 1 (shown in Fig. 12):

1) Reuse test cases for all common PLUCs
2) Instantiate test cases for DispatchingPolicy PLUC from Domain Engineering ✓
3) Instantiate test cases for WrapUpPolicy PLUC from Domain Engineering ✓
4) Instantiate test cases for QueuingPolicy PLUC from Domain Engineering ✓
5) Instantiate test cases for TopologyOfSS PLUC from Domain Engineering ✓

- Product 2 (shown in Fig. 13):

1) Reuse test cases for all common PLUCs
2) Reuse test cases for DispatchingPolicy PLUC from Product 1
3) Reuse test cases for WrapUpPolicy PLUC from Product 1
4) Instantiate test cases for QueuingPolicy PLUC from Domain Engineering ✓
5) Instantiate test cases for TopologyOfSS PLUC from Domain Engineering ✓

- Product 3 (shown in Fig. 14):

1) Reuse test cases for all common PLUCs
2) Reuse test cases for QueuingPolicy PLUC from Product 2
3) Reuse test cases for TopologyOfSS PLUC from Product 1
4) Instantiate test cases for DispatchingPolicy PLUC from Domain Engineering ✓
5) Instantiate test cases for WrapUpPolicy PLUC from Domain Engineering ✓

Table 2 shows the results of comparing the three strategies of SPL testing.

Table 2. Comparison between Different Strategies

Strategy	Generation %	Reuse %	Instantiation %	Phase
Product by Product	100 %	0 %	0 %	Application Engineering
Incremental Strategy	60 %	40 %	0 %	Application Engineering
Reusable Asset Instantiation	33.3 %	26.7 %	40 %	Domain Engineering & Application Engineering

7. Conclusion

In this paper, we have considered a sample product line, which we have designed, implemented then

tested; we have no theory to validate and no process to promote, rather the purpose of our experiment is to explore what comes naturally, against the background of what we understand to be the general obstacles to testing product lines.

- In the domain engineering phase, we have found it advantageous to focus our testing effort on those components that are least affected by variabilities, and that lend themselves best to formal specifications [7], [8]. As far as generating test data, we found it natural to generate random data for which component behavior can be predicted statistically (as is the case for component Arrivals) or computed deterministically (as is the case for component QS). Also, we were able to generate, at low cost, effective test drivers that simulate the use of the component in target applications. As a sign of the effectiveness of the method, we were able to find a fault in the QS component because its behavior deviated originally from the (deterministic) expected behavior.
- In application engineering, the focus of our testing shifted from the low-variability components (which were checked at domain engineering) to the high-variability components (in our case, the main program). This task partakes on integration testing [9] and on regression testing, in the sense that it focuses on the interaction between components rather than individual component behavior, and in the sense that it assumes the correctness of some parts of the product to raise suspicion about other parts. Due to the random nature of the simulation application, we were unable to validate the behavior of the application against precisely defined criteria, but rather to validate it against statistical predictions.

The sample product line discussed in this paper may be a practical case study for the analysis of testing strategies in the analysis of software product lines; it offers a wide range of variabilities, while at the same time being easy to model and specify.

Acknowledgment

We would like to thank Dr. Abdelrahman Osman Elfaki.

References

[1] Pohl, K., & Metzger, A. (2006). Software product line testing. *Communications of the ACM-Software Product Line*, *49(12)*, 78-81.

[2] Mili, H., Mili, A., Yacoub, S., & Addy, E. (2002). *Reuse Based Software Engineering: Techniques, Organization and Measurement.* New York: John Wiley and Sons.

[3] Weiss, D. M., & Lai, C. T. R. (1999). *Software Product Line Engineering: A Family Based Software Development Process* (1st ed.). Reading, Mass.: Addison Wesley.

[4] Colanzi, Elita, T., *et al*. (2013). Evaluating different strategies for testing software product lines. *Journal of Electronic Testing*, *29(1)*, 9-24.

[5] Bertolino, A., & Gnesi, S. (2003). Use case-based testing of product lines. *Proceedings of the SIGSOFT Software Engineering, Vol. 28* (pp. 355–358). New York, NY, USA.

[6] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* (Technical Report CMU/SEI-90-TR-021). SEI, Carnegie Mellon University.

[7] Kahsai, T., Roggenbach, M., & Schlingloff, B. H. (2008). Specification based testing of software product lines. *Proceedings of the Sixth IEEE International Conference on Software Engineering and Formal Methods* (pp. 149-158). Cape Town, South Africa.

[8] Ali, M. M., & Moawad, R. (2010). An approach for requirement based software product line testing. *Proceedings of the Seventh International Conference on Informatics and Systems* (pp. 1-10). Cairo, Egypt.

[9] Machado, C., Neto, P. A. da M. S., & Almeida, E. S. de, (2012). Towards an integration testing approach for

software product lines. *Proceedings of the Thirteenth IEEE International Conference on Information Reuse and Integration* (pp. 616-623). Las Vegas, NV.

Sara Tawfig Osman is a lecture of computer science at the Sudan University of Science and Technology in Khartoum, Sudan. She holds a MSc. degree from the Sudan University of Science and Technology in Khartoum, Sudan. Her research interests are in software engineering.

Ali Mili is a professor of computer science at the New Jersey Institute of Technology in Neawrk, NJ. He holds a PhD degree from the University of Illinois and a doctorat d'etat from the University of Grenoble. His research interests are in software engineering.