Divide and Conquer

Optimizing the release policy of software versions

Philippe Cara[1], Isaac Corro Ramos[2,3], Tammo Jan Dijkema[4], Lusine Hakobyan[3], Markus Heydenreich[2,3], Ellen Jochemsz[3], Tim Mussche[3], Mihaela Popoviciu-Draisma[5], Jacques Resing[3]

Abstract

In this paper we try to find the optimal number of partitions to be made in a piece of software. A model is made for the time-to-market, with respect to which this number is optimized. Refinements are made in this model, taking into account capacity constraints and waiting times. Also, a suggestion is made to use pairwise testing.

Keywords: software partitioning, release policy, time-to-market, testing strategy

3.1 Introduction

ASML, located in Veldhoven, is one of the world's largest producers of lithography systems. Its customers are chip manufacturers, including large companies such as Intel. The chip market is a market with very specific demands. In these times of rapid technological development, it is extremely important to be fast in following new developments on the market. Being the first to offer some feature gives ASML a large advantage over its competitors.

The problem that we are presented with comes from the software department of ASML. They want to keep the software of all machines up to date (the software is such that all machines run on the same software). Currently, ASML issues about 3 new releases of the entire software per year, each with a time-to-market (TTM) of 9 months. Here, time-to-market is defined as the time that elapses between the decision of making a new software release and issuing the tested software to the market. Such a monolithical release includes

1: Vrije Universiteit Brussel, 2: EURANDOM, 3: Technische Universiteit Eindhoven, 4: Universiteit Utrecht, 5: Universität Basel

both bug fixes and new features (Some bug fixes are also delivered in the form of patches, which can be applied cheaper. However, we will not consider this form of updating in this paper.) Installing a new release is very costly for the machine owners. Bringing a machine down could easily cost thousands of dollars per hour. Therefore, some customers choose not to install a new release if there is no urgent reason for it. ASML still supports all older releases.

This form of updating is undesirable for some clients. Suppose a client wishes one new feature. It requests the feature to ASML, which will start implementing it. The update will only be possible in the next release of the entire software, about 9 months away. Also, when the new software is issued to the customer, it comes with all kinds of other features—and possibly bugs.

An alternative is to split the software into a number of pieces, which we will call modules (we were asked to assume this is possible, see e.g. [2]). If a new feature is limited to one module, clients wishing this feature can immediately install the new module once it is released. Other clients can wait longer and install multiple modules at once at a convenient time.

A disadvantage for the software department is that they have to test the new module in a number of environments. Some customers will have the latest version of all other modules installed, but others may still have an old version of another module. Simply requiring all customers to have the latest version of everything is not an option here. What we will require is that all customers have some recent version of all modules (where 'recent' will mean something like 'at most one year old').

It is easy to see that this approach will lead to an increased testing effort; in principle the number of tests will grow exponentially with the number of modules. However, some customers are happier, because the new feature will be available to them earlier.

The question ASML asks is:

> What is the optimal number of modules to split the software into, such that the time-to-market is minimal, while the amount of work remains below some upper bound?

In this paper, we focus on various aspects of this problem. In Section 2, our general model is defined. Next we look at some computational results in this model in the case without capacity constraint in Section 3. In Section 4 we extend our model to take into account a certain capacity of the company that cannot be exceeded. In Section 5 we consider a model including waiting times. Finally we look at some ways of pairwise testing to reduce the cost of testing a new module against older versions of other modules in Section 6. We give some concluding remarks in Section 7.

3.2 The general model

The decision to update a software module will be made by the management based on customer requests. Possibly, by the time of the decision, all development resources are already in use and the development of the new release is

delayed. However, except for Section 3.5, we will assume that the waiting time is 0. This seems reasonable since when a decision to update a software module is made, the current workload of the development resources can be taken into account.

Our aim is to derive a model for the time-to-market when splitting the monolithic software into k pieces. First we want to introduce and discuss the model parameters on a general level, and later make assumptions about these parameters and understand how they influence the outcome.

The proportional size c_j of a module. We want to analyse how the mean time-to-market of the software modules behaves, if we split up the software into k modules. The proportional size of module $j \in \{1, \ldots, k\}$ is denoted by c_j, where $\sum_{j=1}^{k} c_j = 1$. We leave open the meaning of size, one could take, e.g., the number of functionalities.

The development time d_j of a module. We denote the development time of the monolith by D. After splitting the monolith into k modules, the development time of a module is modelled proportional to the size of the module. So we have

$$d_j = Dc_j.$$

The testing time t_j of a module. We denote by T the testing time of the monolith. A new version of a module j will need two kinds of tests prior to the release. The first one will check all the new features of the module in combination with the latest versions of the other $k-1$ modules. The duration of this test is proportional to the size of module j. A second test will verify whether the new version of module j is compatible with all supported versions of the other modules, except the configuration tested previously. The duration of a such test is denoted by A. Thus

$$t_j = Tc_j + A\left(\prod_{i \neq j} l_i - 1\right),$$

where l_i is the number of supported versions of block i.

For the time-to-market TTM_j of a module j we thus have

$$\mathrm{TTM}_j = Dc_j + Tc_j + A\left(\prod_{i \neq j} l_i - 1\right).$$

The number of supported versions l_j of a module. During the compatibility test, ASML tests whether the new release of a software module is compatible with the last l_j versions of the other modules. Hereby l_j is chosen such that all software issued in the last year is supported. Denoting the number of module releases per year by r and writing f_j for the probability that a randomly chosen update request concerns the j-th module, we obtain

$$l_j = \mathrm{Max}\{1, f_j\, r\}.$$

We imposed the restriction $l_j \geq 1$, because we want to support at least the newest configuration for every software module, even if it is not updated every year.

Thus we finally obtain

$$\mathbb{E}\left(\mathrm{TTM}(k)\right) = \sum_{j=1}^{k} f_j \left(Dc_j + Tc_j + A \left(\prod_{i \neq j} \mathrm{Max}\{1, f_i\, r\} - 1 \right) \right) \tag{3.1}$$

for the expected time-to-market with k modules.

3.3 A model without capacity restrictions

Choosing model parameters

ASML plans to split the monolithic software into modules of about the same size. Thus, $c_j = 1/k$ for all modules j. The development time of a module j will then be $d_j = D/k$, and the testing time of a new version of a module, in which all the new features of the module are checked, will be T/k. Thus (3.1) simplifies to

$$\mathbb{E}\left(\mathrm{TTM}(k)\right) = \frac{D}{k} + \frac{T}{k} + A \sum_{j=1}^{k} f_j \left(\prod_{i \neq j} \mathrm{Max}\{1, f_i\, r\} - 1 \right). \tag{3.2}$$

Based on the experience from the monolithic approach, we further assume the following:

- The development time of the monolith is $D = 180$ days.
- The testing time of the monolith is $T = 70$ days.
- A compatibility test needs $A = 2$ days.

The mean time-to-market for different values of k depends on the number of updated modules per year r and on the proportion f_j of update requests that goes to module j. Note that both r and f_j depend on k. Depending on the choice of these functions, the mean time-to-market may change significantly. We will provide calculations for specific choices of these parameters. However, these assumptions need to be checked carefully when validating the model.

The number $r = r(k)$ of module releases per year. Here we write $r(k)$ instead of r to emphasize the k-dependence. Currently the company releases each year about 3 new versions of the monolith. As a first guess, the linear function

$$r(k) = 3k \tag{3.3}$$

seems a good candidate. However, it is rather unclear whether a new version of the monolith would give new features to each of its k submodules. In particular,

due to the extra work needed for the compatibility tests, we expect $r(k)$ to behave sublinear. Nevertheless, (3.3) provides a useful upper bound.

From discussions with ASML representatives we understood that $r(k) = 3k$ might be realistic though. To understand the impact of this choice, we contrast (3.3) with the concave function

$$r(k) = 3k^{\beta}, \tag{3.4}$$

where $0 < \beta < 1$.

The proportion f_j of update requests that go to module j. The request probabilities for different modules of the monolith are unknown, though it is expected to be rather uneven distributed. That makes it hard to find a pertinent probability distribution for our model. We assume here that the update requests for the k modules are distributed according to the Zipf's law, i.e., a module j will be requested by the customers with the probability f_j, where

$$f_j = \frac{j^{-\alpha}}{\sum_{i=1}^{k} i^{-\alpha}}, \tag{3.5}$$

in which we take $\alpha = 0.7$. Zipf's law is observed in many applications, e.g. access of web pages or keyword usage in a search engine. For more information on modeling internet traffic using Zipf's law, including technical aspects, we refer to Cunha et al. [7]. The value of $\alpha = 0.7$ as a model for web requests has been suggested by Breslau et al. [4]. Interestingly, Zipf's law was originally used as a model in philology [13].

Computational results

Case $r = 3k$. Using equation (3.2) together with (3.3) and (3.5), we obtain for $r = 3k$ and $\alpha = 0.7$ that $\mathbb{E}\,(\mathrm{TTM}(k))$ achieves its minimal value for $k = 3$:

$$\mathbb{E}\,(\mathrm{TTM}(3)) = 96.7.$$

If we split the monolith into 3 modules, the mean time-to-market for the release of one module would be 96.7 days.

Here we have that

- the most popular module would have 4.3 new versions per year;
- the second one would have 2.6 new versions per year;
- the last one would have 2 versions per year.

Comparing $96.7 \times 3 = 290.1$ with $D + T = 250$, we see that the splitting requests a supplementary volume of work equivalent to approximately 40 days. These are the compatibility tests. Depending on which module is updated, there will be necessary maximum 11, respectively minimum 5 compatibility tests. To overcome the problem, the testing resources could be extended, or a model with capacity restrictions as in Section 3.4 could be considered.

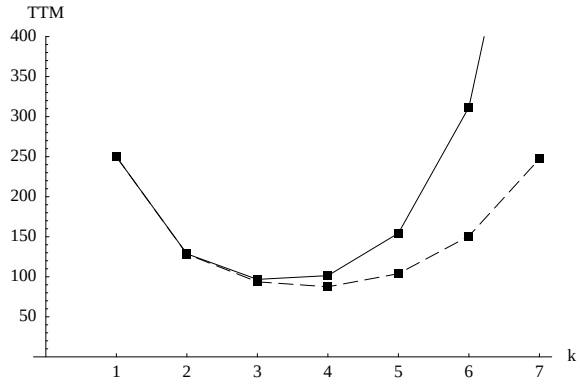

Figure 3.1: The mean time-to-market $\mathbb{E}(\mathrm{TTM}(k))$ in days for $\alpha = 0.7$, and $\beta = 1$ (solid) or $\beta = 0.9$ (dashed).

Case $r = 3k^{0.9}$. In the case $\beta = 0.9$, the minimal time-to-market is achieved for $k = 4$ and

$$\mathbb{E}(\mathrm{TTM}(4)) = 87.5.$$

The mean time-to-market for a module would be 87.5 days. Now

- the most popular module would have 4.2 new versions/year;
- the second one would have 2.6 new versions/year;
- the last two would have 2, respectively 1.6 versions/year.

The supplementary volume of work would be equivalent to about 12 days, but the number of cross tests increases considerably: depending on which module is updated, there are maximum 23, respectively minimum 11, cross tests necessary.

3.4 A model with capacity constraint

Theory

In the last section, we have not yet taken into account the capacity constraint of ASML. Let us assume that the total effort of developing and testing the different modules must stay within the current capacity of the company. The current capacity can be taken as 3 times the total time needed for one new release of the monolith (since at the moment, all ASML's machines and people are working on 3 releases of the full software program per year). This means that the available capacity is

$$\mathrm{cap} = 3(D + T).$$

As before, we assume that we know f_j, the proportion of feature requests for module j, and r, the total number of feature requests per year. So to make every customer happy, we should have a new release each time there is a request. This would be $f_j r$ per year for module j. However, to stay within our capacity the management should decide to put an upper limit M on the number of releases for one module. This means that, in case there are many requests for a certain module, we won't release new modules at each request but rather have M releases of that module per year. So we put

$$\begin{aligned} r_j &:= \#\text{ releases of module } j \text{ per year} = \operatorname{Min}\{f_j r, M\}, \\ \widetilde{f}_j &:= \text{proportion of releases of module } j = \frac{r_j}{\sum_{i=1}^{k} r_i}, \\ M &:= \text{maximal number of releases of each module per year.} \end{aligned}$$

We can now divide our modules into three groups, namely very popular modules, medium popular modules and least popular modules (with respect to feature requests). If we order the modules according to r_j (from high number of releases to low number of releases), we get

Module:	1	...	m	$m+1$	...	n	$n+1$	...	k
$f_j r \in$	$(3,\infty)$	...	$(3,\infty)$	$(1,3]$	...	$(1,3]$	$(0,1]$	...	$(0,1]$
$r_j =$	M	...	M	$f_j r$	...	$f_j r$	$f_j r$	...	$f_j r$
$l_j =$	r_j	...	r_j	r_j	...	r_j	1	...	1

The idea behind this is to release as many versions of the less and medium popular modules as are requested (which per module is at most the current 3 releases per year), and spend the time that is left on releasing M versions of each popular module per year. One can immediately see that this can only give an advantage to the current approach when $M > 3$ is within reach.

So let us compute for a given k, the maximal M to satisfy the capacity constraint. The total time needed to develop and test a new release of module j is given by

$$\mathrm{TTM}_j = \frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} l_i - 1 \right),$$

where A is the (small) testing time needed to test the new version of module j against the previous versions of all modules. Since the total time spent on the r_j releases of module j per year is $r_j \,\mathrm{TTM}_j$, the total time that we spend on all releases of all modules per year is

$$\sum_{j=1}^{k} r_j \cdot \left(\frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} l_i - 1 \right) \right).$$

Notice that M appears in the product $\prod_{i \neq j} l_i$. Hence, for a given k, we can find the maximal value of M such that

$$\sum_{j=1}^{k} r_j \cdot \left(\frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} l_i - 1 \right) \right) < 3(D+T).$$

For this optimal value of M, we can determine the mean time-to-market for a module

$$\mathbb{E}\left(\mathrm{TTM}(k)\right) = \sum_{j=1}^{k} \widetilde{f}_j \cdot \left(\frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} l_i - 1 \right) \right),$$

and compare these for the different values of k to see which choice for k is best.

Examples

In practice, the outcome of the analysis will, of course, depend on the constants D, T, and A, for which we will for now substitute $D = 180$, $T = 70$ and $A = 2$. But most importantly, it will depend on the 'popularity rate' f_j of the modules, which ASML should evaluate thoroughly before making a decision. The two examples that we consider in this section are

- The f_j are distributed according to Zipf's law, and r (the total number of requests when splitting the program into k parts), is considered to be $r = 3k$:

$$f_j r = \frac{j^{-\alpha}}{\sum_{i=1}^{k} i^{-\alpha}} \cdot 3k, \text{ where } \alpha = 0.7 \text{ (see Section 3.3).}$$

- The f_j are distributed according to a 'toy example', that originates from the fact that for $k = 5$, ASML can give a guess for a suitable approximation of $f_j r$:

$$f_1 r = 12,\ f_2 r = 12,\ f_3 r = 1,\ f_4 r = \frac{1}{2},\ f_5 r = \frac{1}{3}.$$

In the first example, we have

k	$f_1 r$	$f_2 r$	$f_3 r$	$f_4 r$	$f_5 r$	$f_6 r$	$f_7 r$	$f_8 r$	$f_9 r$	$f_{10} r$
1	3	–	–	–	–	–	–	–	–	–
2	3.71	2.29	–	–	–	–	–	–	–	–
3	4.33	2.66	2.01	–	–	–	–	–	–	–
4	4.88	3.01	2.26	1.85	–	–	–	–	–	–
5	5.39	3.32	2.50	2.04	1.75	–	–	–	–	–
6	5.87	3.61	2.72	2.22	1.90	1.67	–	–	–	–
7	6.32	3.89	2.93	2.29	2.05	1.80	1.62	–	–	–
8	6.75	4.15	3.13	2.56	2.19	1.93	1.73	1.57	–	–
9	7.16	4.41	3.32	2.71	2.32	2.04	1.83	1.67	1.54	–
10	7.55	4.65	3.50	2.86	2.45	2.16	1.93	1.76	1.62	1.51

As explained, we will divide the modules for a given k into three groups, the very/medium/least popular modules. The first group, namely the one for which $f_i r > 3$, consists of typically one, two or three modules. For these most popular modules, we will fix the number of releases per year at M, so the expected number of releases of the modules will be

k	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
1	M	–	–	–	–	–	–	–	–	–
2	M	2.29	–	–	–	–	–	–	–	–
3	M	2.66	2.01	–	–	–	–	–	–	–
4	M	M	2.26	1.85	–	–	–	–	–	–
5	M	M	2.50	2.04	1.75	–	–	–	–	–
6	M	M	2.72	2.22	1.90	1.67	–	–	–	–
7	M	M	2.93	2.29	2.05	1.80	1.62	–	–	–
8	M	M	M	2.56	2.19	1.93	1.73	1.57	–	–
9	M	M	M	2.71	2.32	2.04	1.83	1.67	1.54	–
10	M	M	M	2.86	2.45	2.16	1.93	1.76	1.62	1.51

Assuming $l_j = \text{Max}\{1, r_j\}$ we can determine the maximal M for each k to satisfy

$$\sum_{j=1}^{k} r_j \cdot \left(\frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} l_i - 1 \right) \right) < 3(D+T).$$

We obtain the following maximal values of M:

k	2	3	4	5	6	7	8	9	10
M	3.55	3.26	2.46	1.77	1.17	0.76	0.93	0.71	0.53

One can see that only $k = 2$ or $k = 3$ might be an improvement on the current $k = 1$. Suppose we split in two modules, and we schedule 3.50 releases of the most popular module per year, and 2.29 releases of the least popular module.

Then the mean time-to-market is

$$\begin{aligned}
\mathbb{E}\left(\text{TTM}(2)\right) &= \sum_{j=1}^{k} \widetilde{f}_j \cdot \left(\frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} l_i - 1 \right) \right) \\
&= \sum_{j=1}^{2} \frac{r_j}{\sum_{i=1}^{k} r_i} \cdot \left(\frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} \text{Max}\{1, r_i\} - 1 \right) \right) \\
&= \frac{3.50}{5.79} \cdot \left(\frac{70}{2} + \frac{180}{2} + 2 \cdot (2.29 - 1) \right) \\
&\quad + \frac{2.29}{5.79} \cdot \left(\frac{70}{2} + \frac{180}{2} + 2 \cdot (3.55 - 1) \right) \approx 129 \text{ days.}
\end{aligned}$$

So, while staying within the current capacity, it is possible to split into two modules such that the time that elapses after the management has asked for an update of a module is about 4 months on average.

If we split in three modules, we find

$$\begin{aligned}\mathbb{E}\left(\mathrm{TTM}(3)\right) &= \sum_{j=1}^{3} \frac{r_j}{\sum_{i=1}^{k} r_i} \cdot \left(\frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} \mathrm{Max}\{1, r_i\} - 1 \right) \right) \\ &= \frac{3.26}{7.93} \cdot \left(\frac{70}{3} + \frac{180}{3} + 2 \cdot (2.01 \cdot 2.66 - 1) \right) \\ &\quad + \frac{2.66}{7.93} \cdot \left(\frac{70}{3} + \frac{180}{3} + 2 \cdot (2.01 \cdot 3.26 - 1) \right) \\ &\quad + \frac{2.01}{7.93} \cdot \left(\frac{70}{3} + \frac{180}{3} + 2 \cdot (2.66 \cdot 3.26 - 1) \right) \approx 95 \text{ days.}\end{aligned}$$

We conclude that for a popularity rate that is Zipf-distributed, $k = 3$ is optimal, as in the previous chapter where the capacity constraint was not taken into account.

Now let us take a look at the second example, the so-called 'toy example'. Recall that this is the case where the popularity rate f_j of the modules is not distributed according to Zipf's law, but that we have

$$f_1 r = 12,\ f_2 r = 12,\ f_3 r = 1,\ f_4 r = \frac{1}{2},\ f_5 r = \frac{1}{3}.$$

In other words, we assume that ASML can create 5 modules of approximately the same size, such that one has to be changed once every three years, one has to be changed once every two years, one has to be changed once a year, and the two most popular modules need changes every month.

Now that we have a good approximation of the number of requests in the case that ASML splits the monolith into five parts, can we use that to conclude something in comparison with other values of k? It seems logical to assume that for $k = 2$, it is possible to create one module that has to be changed once a year, and one module that still has to be changed 12 times a year (because to make two modules out of the five proposed by ASML, we would take the first two together with a part of the third so there will be requests to change this big module once every year). For $k = 3$, $k = 4$, or $k = 6$, it is harder to say something reasonable, because we cannot guess how the number of changes per year for each module would be distributed. So let us compare $k = 1$, $k = 2$, and $k = 5$ for this 'toy example':

k	r_1	r_2	r_3	r_4	r_5
1	3	–	–	–	–
2	M	1	–	–	–
5	M	M	1	0.50	0.33

As before, we need to find the maximal values of M such that

$$\sum_{j=1}^{k} r_j \cdot \left(\frac{D}{k} + \frac{T}{k} + A \cdot \left(\prod_{i \neq j} \mathrm{Max}\{1, r_i\} - 1 \right) \right) < 3(D + T).$$

We obtain that:

k	2	5
M	4.93	6.22

It follows that

$$\begin{aligned}
\mathbb{E}\left(\mathrm{TTM}(2)\right) &= \frac{4.93}{5.93}\cdot\left(\frac{70}{2}+\frac{180}{2}+2\cdot(1-1)\right)\\
&\quad+\frac{1}{5.93}\cdot\left(\frac{70}{2}+\frac{180}{2}+2\cdot(4.93-1)\right)\approx 126 \text{ days},\\
\mathbb{E}\left(\mathrm{TTM}(5)\right) &= \frac{6.22+6.22}{14.27}\cdot\left(\frac{70}{5}+\frac{180}{5}+2\cdot(1\cdot 1\cdot 1\cdot 6.22-1)\right)\\
&\quad+\frac{1+\frac{1}{2}+\frac{1}{3}}{14.27}\cdot\left(\frac{70}{5}+\frac{180}{5}+2\cdot(1\cdot 1\cdot 6.22\cdot 6.22-1)\right)\\
&\approx 69 \text{ days}.
\end{aligned}$$

One can clearly see that for this example, splitting into five modules gives the best results.

We have shown in this section that splitting into modules while staying within the capacity is possible and can result in a shorter mean time-to-market. However, it is essential for the validity of the results to have reliable information on the distribution of the number of requests over the modules.

3.5 A model including waiting times

In this section we introduce a queueing model to study the mean time-to-market of releases of modules. Assume that the number of modules in which we divide the monolith is equal to k. The model is a closed queueing network with two stations, one consisting of k parallel servers and one consisting of a single server and a request queue (see Figure 3.2).

The first station represents the modules for which no new releases are requested. The second station represents the modules for which a new release is requested. After the release of a new version of module i, the next request for a release of module i occurs after an exponentially distributed time with parameter λ_i, $i = 1, \ldots, k$. Here, $1/\lambda_i$ is the mean time until the next request for module i occurs. Typically, the $\lambda_i's$ are different because not all the modules have the same rate of being requested for a new release since there are modules that are more popular than others. Modules requested for a new release queue up in the request queue until they can be served. The server in the second station represents the group of approximately 400 employees working on the modules. We assume that the server works in a processor sharing fashion. Whenever there are j requests in the request queue the server splits its capacity equally over the j requests. The service time of module i in the second station is exponentially distributed with parameter μ_i, $i = 1, \ldots, k$. Here $1/\mu_i$ is the

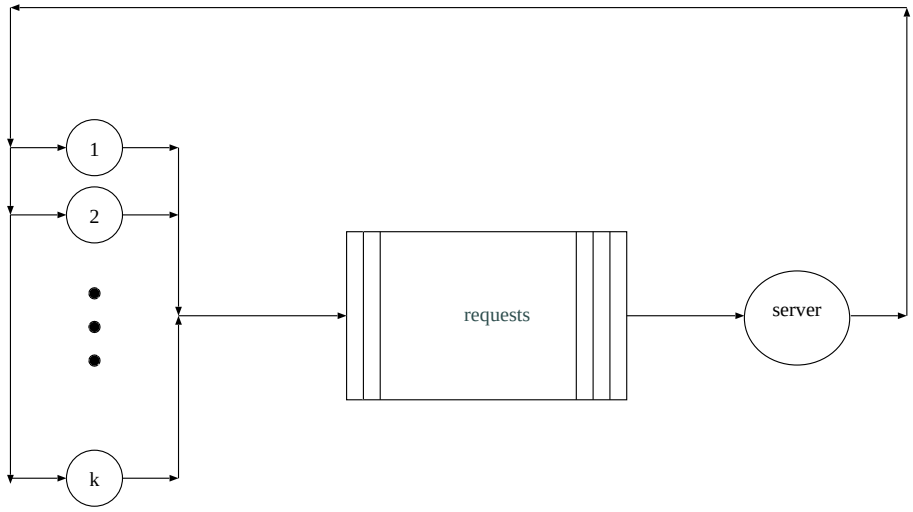

Figure 3.2: The closed queueing network

mean time a request would spend in the second station whenever there would be no other requests at the same time at this station.

The random variable $X_t^{(i)}, i = 1, \ldots, k$, denotes the state of module i at time t, i.e.,

$$X_t^{(i)} = \begin{cases} 0 & \text{if a new release for module } i \text{ is requested at time } t, \\ \\ 1 & \text{otherwise.} \end{cases}$$

The stochastic process $X(t) = (X_t^{(1)}, X_t^{(2)}, \ldots, X_t^{(k)})$ is a continuous-time Markov process. The equilibrium distribution of this continuous-time Markov process can be obtained by solving the balance equations, equating the inflow and outflow of each state, together with the normalization equation. This is illustrated by the following example.

Example 3.5.1. *Let us consider the case $k = 3$. We write p_{i_1,i_2,i_3}, where $i_1, i_2, i_3 \in \{0, 1\}$, to denote the equilibrium probability of the system to be in the state (i_1, i_2, i_3). For example $p_{0,0,0}$ is the probability that new releases for all modules are requested and $p_{1,1,1}$ is the probability that no new releases are requested. The balance equations are given by*

$$
\begin{aligned}
\left(\frac{\mu_1}{3}+\frac{\mu_2}{3}+\frac{\mu_3}{3}\right) p_{0,0,0} &= \lambda_1\, p_{1,0,0}+\lambda_2\, p_{0,1,0}+\lambda_3\, p_{0,0,1},\\
\left(\frac{\mu_2}{2}+\lambda_1+\frac{\mu_3}{2}\right) p_{1,0,0} &= \frac{\mu_1}{3}\, p_{0,0,0}+\lambda_2\, p_{1,1,0}+\lambda_3\, p_{1,0,1},\\
\left(\frac{\mu_1}{2}+\lambda_2+\frac{\mu_3}{2}\right) p_{0,1,0} &= \frac{\mu_2}{3}\, p_{0,0,0}+\lambda_1\, p_{1,1,0}+\lambda_3\, p_{1,0,1},\\
\left(\frac{\mu_1}{2}+\lambda_3+\frac{\mu_2}{2}\right) p_{0,0,1} &= \frac{\mu_3}{2}\, p_{0,0,0}+\lambda_1\, p_{1,0,1}+\lambda_3\, p_{0,1,1},\\
(\mu_3+\lambda_1+\lambda_2)\, p_{1,1,0} &= \frac{\mu_2}{2}\, p_{1,0,0}+\frac{\mu_1}{2}\, p_{0,1,0}+\lambda_3\, p_{1,1,1},\\
(\mu_2+\lambda_1+\lambda_3)\, p_{1,0,1} &= \frac{\mu_3}{2}\, p_{0,1,0}+\frac{\mu_1}{2}\, p_{0,0,1}+\lambda_2\, p_{1,1,1},\\
(\mu_1+\lambda_2+\lambda_3)\, p_{1,1,1} &= \frac{\mu_3}{2}\, p_{0,1,0}+\frac{\mu_2}{2}\, p_{0,0,1}+\lambda_1\, p_{1,1,1},\\
(\lambda_1+\lambda_2+\lambda_3)\, p_{1,1,1} &= \mu_3\, p_{1,1,0}+\mu_2\, p_{1,0,1}+\mu_1\, p_{0,1,1},
\end{aligned}
$$

and the normalization equation is

$$p_{0,0,0}+p_{1,0,0}+p_{0,1,0}+p_{0,0,1}+p_{1,1,0}+p_{1,0,1}+p_{0,1,1}+p_{1,1,1}=1.$$

Solving the above system of equations we obtain

$$
\begin{aligned}
&p_{0,0,0}=6C\lambda_1\lambda_2\lambda_3,\ p_{1,0,0}=2C\lambda_2\lambda_3\mu_1,\ p_{0,1,0}=2C\lambda_1\lambda_3\mu_2,\\
&p_{0,0,1}=2C\lambda_1\lambda_2\mu_3,\ p_{0,1,1}=C\lambda_1\mu_2\mu_3,\ p_{1,1,0}=C\lambda_3\mu_1\mu_2,\\
&p_{1,0,1}=C\lambda_2\mu_1\mu_3,\ p_{1,1,1}=C\mu_1\mu_2\mu_3.
\end{aligned}
$$

where

$$
\begin{aligned}
\frac{1}{C} &= 6\lambda_1\lambda_2\lambda_3+2\lambda_2\lambda_3\mu_1+2\lambda_1\lambda_3\mu_2+2\lambda_1\lambda_2\mu_3\\
&\quad +\lambda_3\mu_1\mu_2+\lambda_2\mu_1\mu_3+\lambda_1\mu_2\mu_3+\mu_1\mu_2\mu_3.
\end{aligned}
$$

In the case of an arbitrary number of modules we can also obtain a closed expression for the equilibrium distribution, see [1]. The equilibrium probabilities are given by

$$p_{i_1,i_2,\ldots,i_k}=C\cdot\left(k-\sum_{j=1}^{k} i_j\right)!\cdot\prod_{j=1}^{k}\left(\lambda_j^{1-i_j}\cdot\mu_j^{i_j}\right).$$

where C is chosen such that the sum of the probabilities equals one.

Once we know the equilibrium distribution, we can obtain other performance measures for the system. We denote by $\mathbb{E}\,(\mathrm{L}(k))$ the mean number of modules in the request queue and by $\mathbb{E}\,(\mathrm{TTM}(k))$ the mean time-to-market for an arbitrary module, i.e., the time between the instant that the release is requested and the instant the new version of the module is released. We can

easily relate these two measures using Little's formula, see e.g. [10]. If $\delta(k)$ is the rate at which new releases for modules are requested, Little's formula gives $\mathbb{E}(\mathrm{L}(k)) = \delta(k)\mathbb{E}(\mathrm{TTM}(k))$. The mean number of modules in the request queue and the rate at which new releases for modules are requested can be calculated using

$$\begin{aligned} \mathbb{E}(\mathrm{L}(k)) &= \sum_{i \in I} p_{i_1,i_2,\ldots,i_k} \cdot \left(k - \sum_{j=1}^{k} i_j \right), \\ \delta(k) &= \sum_{i \in I} p_{i_1,i_2,\ldots,i_k} \cdot \left(\sum_{j=1}^{k} (i_j \cdot \lambda_j) \right) \end{aligned}$$

with

$$I = \{i = (i_1, \ldots, i_k) : i_j \in \{0, 1\} \text{ for all } j\}.$$

The mean time-to-market finally follows from Little's formula.

In the model described in this section, for each module only one request for a new release can be in the request queue. If two or more requests for a module can be simultaneously in the request queue, the model should be adapted. When there can be at most a fixed number of requests for a module simultaneously in the request queue, this can be included in the model by increasing the number of modules in the closed queueing network (e.g. from k to $3k$ if there are at most 3 requests for a module simultaneously in the request queue). If the number of requests simultaneously in the request queue for a certain module is unlimited, probably an open queueing model instead of a closed queueing model is more appropriate. For these open models also results are available for the mean time jobs spend in the system (see e.g. [11] for a formula for the mean time a job spends in the system in an $M/G/1$ processor sharing queue).

3.6 Pairwise Testing

In this section we introduce a method for reducing the number of test cases and therefore for reducing the test effort. We are not interested in functional unit testing but in cross testing, i.e., testing different versions of modules against each other. The testing effort depends on the number of versions of the other modules because a new version of a module should be tested with all combinations of all versions of the other modules. This testing method is known as exhaustive testing. This way of testing covers all test cases. Due to its high cost, to accomplish exhaustive testing in practice is in most cases not feasible. In contrast to exhaustive testing, pairwise testing is only covering all pairwise combinations of versions of modules. This means that for any two modules M_1 and M_2 and any two versions V_1 of M_1 and V_2 of M_2, there is a test in which M_1 has version V_1 and M_2 has version V_2.

Different test generation strategies have been published for pairwise testing. Here we briefly describe three of them. In the first approach, if all the pairs in a given combination exist in other combinations we drop that combination, see [3]. Table 3.1 shows the test cases if we consider to divide our software into three modules and to support two versions. In practice we should drop the test cases number two and number four since pairwise they exist already. The second case exists in the test cases 3, 5 and 6. The fourth case exists in the test cases 1, 5 and 6.

Test Cases	Module 1	Module 2	Module 3
1	Version 1	Version 2	Version 2
2	Version 2	Version 1	Version 1
3	Version 2	Version 1	Version 2
4	Version 1	Version 2	Version 1
5	Version 2	Version 2	Version 1
6	Version 1	Version 1	Version 1

Table 3.1: Test cases for 3 modules and 2 supported versions using pairwise techniques.

A combinatorial design approach is used by the Automatic Efficient Test Generator (AETG). This strategy requires that every pair is covered at least once. It does not specify how many times each pair is covered. For further details, see [5] and [6]. A third approach is to use orthogonal arrays to generate test cases. Orthogonal arrays are combinatorial designs used to design statistical experiments that require that every pair is covered the same number of times, see [8].

There are many tools available for generating test cases based on pairwise testing. Each of them is using some specific algorithm for generating pairs. The examples shown in this section are generated using a free GUI based tool for generating test cases called CTE-XL. This tool generates the pairs using the Classification-Tree Method which is a testing method used by DaimlerChrysler AG. For further details about the tool, see [12].

Suppose we divide the software into 3 modules and we want to support 3 versions for each of them. Exhaustive testing requires 27 test cases to cover all possible combinations. However using pairwise testing techniques only 9 test scenarios are required, see Table 3.2.

In Tables 3.3 and 3.4 we compare the number of test cases produced using pairwise testing with the number of test cases produced using exhaustive testing. We consider different number of modules and different number of old supported versions. Table 3.3 shows the number of test cases needed using pairwise testing for 2, 3, 4 and 5 modules supporting 2, 3 and 4 old versions respectively. The number of test cases needed using exhaustive testing for 2, 3, 4 and 5 modules supporting 2, 3 and 4 old versions respectively are shown

Test Cases	Module 1	Module 2	Module 3
1	Version 3	Version 2	Version 3
2	Version 1	Version 3	Version 2
3	Version 2	Version 1	Version 1
4	Version 1	Version 1	Version 3
5	Version 2	Version 2	Version 2
6	Version 3	Version 3	Version 1
7	Version 1	Version 2	Version 1
8	Version 2	Version 3	Version 3
9	Version 3	Version 1	Version 2

Table 3.2: Test cases for 3 modules and 3 supported versions using pairwise techniques.

in Table 3.4. Clearly, the number of test cases increases with the number of modules and with the number of supported versions. Furthermore, we see that if we split up the monolith, for example, into four modules supporting four versions the number of test cases using exhaustive testing grows much faster (256) than using pairwise testing (20).

	2 Modules	3 Modules	4 Modules	5 Modules
2 Versions	4	4	5	6
3 Versions	9	9	9	13
4 Versions	16	19	20	23

Table 3.3: Number of test cases using pairwise testing

	2 Modules	3 Modules	4 Modules	5 Modules
2 Versions	4	8	16	32
3 Versions	9	27	81	256
4 Versions	16	64	256	1024

Table 3.4: Number of test cases using exhaustive testing

Of course, it is possible that pairwise testing alone does not detect all bugs. Sometimes they can be found out only by inspecting three or more module interactions. The possible solution could be to complement pairwise testing with another kind of testing or to extend it to all 3-module (or n-module) combinations, but this could also be costly.

Finally let us remark that in the examples we have presented in this section we have assumed for simplicity that we support the same number of versions for each module. In practice this assumption is not always true since we can support different number of versions for every module. In this case we will have some repeated pairs. Another approach, however, could be to use orthogonal arrays to generate the test cases in which all the pairs are covered the same number of times. For further details and applications, see [9].

3.7 Conclusions

We have translated the problem given to us by ASML into a general model that can be extended to include capacity constraint or waiting times. A lot of parameters appear in this model for which a suitable value should be chosen. One of the most important parameters relates to the popularity rate of the different modules which needs to be investigated by ASML to draw the right conclusions. To illustrate the model we worked out a few examples. For this examples it seems that splitting the monolith into a small number of modules can certainly be an improvement. Other techniques, such as pairwise testing, can be used to further reduce the testing time. We should note that once the optimal number of partitions is derived, a lot of work remains to be done. Actually splitting the software into k more or less independent pieces can be very hard. It may be desirable to deviate from the optimal value to make space for natural partitions (such as splitting firmware and user interface).

Acknowledgement. The authors greatly acknowledge stimulating discussions with Tammo van den Berg, Martijn van Noordenburg and Joost Smits from ASML, and with Nebojsa Gvozdenovic (CWI), Małwina Luczak (London School of Economics) and Rob van der Mei (CWI and Vrije Univeriteit Amsterdam).

3.8 Bibliography

[1] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, 1975.

[2] R. Bisseling et al. Partitioning a call graph. In *Proceedings of the Fifty-second European Study Group with Industry, CWI syllabus 55*, pages 95–107, 2006.

[3] M. Bolton. Pairwise testing. `http://www.developsense.com/testing/PairwiseTesting.html`, 2004.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *Proceedings of IEEE Infocom'99*, pages 126–134, New York, March 1999.

[5] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[6] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design apporach to automatic test generation. *IEEE Software*, pages 83–88, 1996.

[7] C. A. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW client-based traces. Technical Report TR-95-010, Boston University Department of Computer Science, April 1995. Revised July 18, 1995.

[8] E. Dustin. Orthogonally speaking. *STQE magazine*, 3(5):46–51, October 2001.

[9] A. S. Hedayat, N. J. A. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*. Springer-Verlag, New York, 1999.

[10] L. Kleinrock. *Queueing Systems, Volume I: Theory*. Wiley Interscience, New York, 1975.

[11] L. Kleinrock. *Queueing Systems, Volume II: Computer Applications*. Wiley Interscience, New York, 1976.

[12] E. Lehmann and J. Wegener. Test case design by means of the CTE XL. In *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review. EuroSTAR 2000*, Kopenhagen, Denmark, December 2000.

[13] G. K. Zipf. Relative frequency as a determinant of phonetic change. *Harvard Studies in Classical Philology*, 40:1–95, 1929.