Visualizing ASML TWINSCAN event log history

H. van Venrooij, BSc

Supervisors
Prof. dr. ir. J.J. van Wijk
TU/e
Dr. ir. P. van den Hamer
ASML

Eindhoven, June 2014
Abstract

ASML’s wafer scanners play a crucial part in the process of making computer chips. They pattern an optical mask of hundreds of chips onto a silicon disc, a so-called wafer. There is no one that knows everything there is to know about a wafer scanner. As such they are black boxes to their engineers. When such a machine fails, there are a number of tools that either analyze the fault automatically or allow an engineer to do it manually. So there is no lack in support for diagnosis.

While dealing with the problem, it is desirable to get a bigger picture of the machine. This would show what happened prior to the fault. Despite this, ASML lacks a tool that allows them to quickly get an idea what a wafer scanner has done over a longer period of time.

This thesis describes an interactive visualization of TWINSCAN event logs. An interactive prototype was designed and built that allows a user to quickly see what a wafer scanner has been doing using familiar concepts. When desired, details of specific periods in time are displayed.
I would like to thank my supervisor at the Eindhoven University of Technology, Prof. dr. ir. J.J. van Wijk, for his guidance during the project. His comments and advice on this thesis and the prototype was sorely needed.

I would also like to thank my supervisor at ASML, Dr. ir. P. van den Hamer, for invaluable information on wafer scanners. The understanding he provided on the actual workings of these machines was sorely needed. Furthermore the supply of feedback on both the prototype as well as this report was more than helpful.

I would also like to thank all remaining colleagues at both the TU/e and ASML for their feedback and any help with bureaucratic non-sense. Otherwise I would still be mucking about without any data. Also the software they provided was extremely useful.

Finally I would like to apologize to my roommates, friends and family for my abhorrent behavior during the last legs of this journey.
# Contents

Abstract .................................................. I

Acknowledgements ........................................ II

Contents .................................................. III

1 Introduction ............................................. 1
   1.1 Objectives ........................................... 1
   1.2 Scope ................................................. 2
   1.3 Definitions .......................................... 2
   1.4 Document structure ................................... 3

2 Problem analysis ......................................... 4
   2.1 Data .................................................. 5
   2.2 Related work ......................................... 11
      2.2.1 Glyph visualizations .............................. 11
      2.2.2 Non-glyph visualizations ......................... 13
      2.2.3 ASML ............................................. 16
      2.2.4 Discussion of related work ....................... 17
   2.3 Missing pieces ........................................ 19
   2.4 Requirements ......................................... 20

3 Log browser ............................................. 23
   3.1 Overview ............................................. 24
   3.2 Productivity view ..................................... 24
   3.3 Activity view ......................................... 29
   3.4 Task and Event view ................................... 32
   3.5 Common functionality ................................. 35

4 Discussion ............................................... 39
   4.1 Scalability ............................................ 41
   4.2 Future work ........................................... 43

5 Conclusions ............................................. 46
Chapter 1

Introduction

If humans are good at one thing, it is making complex artifacts. Excellent examples are ASML’s wafer scanners. These machines play a crucial part in the process of making computer chips. They pattern the design (optical mask) of a number of chips onto a silicon disc, a so-called wafer.

The complexity of these machines is such that the software operating these machines, is described in approximately forty-million lines of code [1]. The cutting edge technologies ASML delivers is not only limited by the complexity of the software, as their techniques progress the limits of physics and mechanics inch ever closer. And over the years, these systems have become so complex that there is not a single ASML employee who knows exactly how all different sub-systems work and interact.

ASML constantly attempts to increase the performance of their wafer scanners, both in terms of throughput (number of wafers per hour) as well as minimize the feature size of chip patterns. One way to increase the performance of the machines is to minimize the time a machine is down due to a part or system failure. The complexity of the machines makes it virtually impossible to identify the cause of issues by examining the external behavior.

To speed up the diagnosis, the progress and failures of wafer scanners are reported in event logs. While these logs are intended to allow people to reason about the behavior of a machine, both the numerous tasks occurring in parallel as well as the sheer volume of the logs, make it tedious at best to manually reason about detailed behavior.

ASML already employs automatic techniques to diagnose errors. But the machines are still partly seen by service engineers as black boxes. This project aims to provide a visual means to show how and what a machine has been doing.

1.1 Objectives

In this thesis, an attempt is made at visualizing:

“How and what has a wafer scanner been doing in the past hours/days?”

in such a way that it can be assessed by an engineer in a matter of seconds.

To do this, we must answer the following questions:
Chapter 1. Introduction

1. What is an event log and how is it structured?

2. What stored information is interesting and to whom?

3. How can this information be presented?

The answer to the first question is presented in Section 2.1, whereas the second question is answered in Section 2.3. The third question is answered in Section 2.2 and Chapter 3.

1.2 Scope

Event logs store a wealth of information, which can be automatically analyzed to suit a wide variety of purposes. This project was intended to provide a generic overview of how and what a wafer scanner was doing for a couple of hours or even days. It was not the goal of this project to do automated analysis, i.e., data or process mining. This means that the data should be presented as-is, or using minimal/trivial transformations at best. Nor should the prototype be meant to be used as means to further detailed diagnostics. On the other hand, within the scope of the project, properties of the functioning of a single wafer scanner (and parts thereof) should be shown.

Finally, the prototype of this project was meant as a prototype, hence it does not have to be production ready. However, the interface and the implementation should at least be understandable such that ASML can incorporate a selection of parts in their own tool chain should the need arise.

1.3 Definitions

The following terms are heavily used within ASML:

**Event:**

The occurrence of “anything” in a wafer scanner. This could be a warning that a sensor measures a value that exceeds its threshold as well as the beginning of a new lot, for instance. Note that most non-error events are not logged to reduce the clutter in the event log.

**Event log:**

An event log is a series of files, which reports on the operation of any manual or automatic process. Whenever the need arises to report on progress, warnings or errors are added to such a file.

**Log record:**

The text that is stored in an event log that represents an event.

**Lot:**

A lot is analogous to a print job on a printer. This contains a number of wafers on which the same optical mask is patterned. A typical lot contains twenty-four wafers.

**Main event:**

Main events are events that are considered to be important enough to be shown at the same level as tasks. An example of such an event could be the loading of a new reticle in the machine, the so-called “pod placed” main event.
Chapter 1. Introduction

**Reticle:**
A reticle is the image of one layer of a chip, which is going to be patterned onto a wafer. This reticle is analogous to a negative in the process of wet photography.

**Task:**
A task is an abstraction from events. These tasks represent a set of events that are related to the same goal. For instance, a lot is a good example of a task that groups all events that relate to the exposure of a set of wafers.

**Wafer:**
A wafer is a silicon disc with a typical diameter of thirty centimeters, which is the basis of a number of chips. Using ultraviolet light, (a part of) the optical mask of a chip is patterned onto this disc (for dozens of different layers).

**Wafer scanner:**
A machine that patterns the optical mask of a number of chips onto a wafer by exposing a layer of photoresist.

1.4 **Document structure**

The remainder of this report is divided into four parts. In Chapter 2 the problem is analyzed by describing what the problem is, what the data looks like (structurally), what already has been done to solve the problems and finally lists several requirements that the prototype has to satisfy. Next in Chapter 3 the prototype is described. The results thereof are discussed in Chapter 4. Finally, the report is concluded in Chapter 5.
Chapter 2

Problem analysis

Event logs are important tools for management purposes. In a textual way they provide information of the history of a system over time. The logged records not only provide information when the system is working properly, but also they indicate when problems occur. Hence these logs can be used to analyze malfunctions and bottlenecks. One of the biggest problems with logs is that they can easily become cluttered with unimportant records, such that critical records are obscured causing problems to go undetected.

Whenever a problem is detected in a wafer scanner, be it in software or hardware, a software engineer analyzes the log in an attempt to find the root cause of the problem. The problem with the manual analysis is that ASML’s wafer scanners produce a log record roughly each second (see also the last paragraph of Section 2.1). Unlike with traditional event logs, progress events aren’t logged often to reduce clutter. So there is no way to easily discard most of the remaining error records because any of them could indicate the root cause of the problem. Furthermore, the sheer complexity of the machine is not exactly helping. The first line support engineers only have a global grasp of the machines. The support engineers are hardly experts on wafer scanners, so for more complex problems help is often needed to solve a problem. All of the above can cause diagnosis to take hours.

Unscheduled down time costs ASML’s customers approximately twenty dollars [2] per second in lost consumer opportunity. To minimize such costs the Diagnostics group develops the leading workflow and the needed tools, for the analysis of these downs as well as failed wafers and other hiccups. Their goal in the long run is to allow support engineers to diagnose over 90% of all problems within an hour. Note that there are a number of niche applications that are not maintained by the Diagnostics group.

Both data mining and visualization techniques attempt to provide insight into large amounts of data. Data mining techniques can be used to extract patterns from large amounts of data. For instance, one could apply clustering techniques to sets of correlated events to see whether a similar problem has occurred previously. If that is the case, one could try the solution for that problem and see whether it solves the current one. Visualization techniques aim to provide a succinct representation of the logged data by graphically showing patterns or by presenting time statistics of records. Hence both data mining and visualization techniques seem to be helpful when analyzing event logs.
But do these techniques actually help with the analysis of event logs? Wongsuphasawat et al. [3] have performed a usability study for their own visualization of event sequences of medical data. The experiment required the users to answer typical questions about real life medical data. During the data exploration process, the users were only allowed to use the specified tool. It was concluded that all but one subject answered all questions correctly within a minute. Furthermore, Wongsuphasawat et al. planted a number of realistic anomalies in the data. They gave their test subjects a small amount of time to identify them, which most of them did successfully. Those that did not find all anomalies, had noted the missing ones, but had discarded them as they did not think that it was extremely abnormal behavior. Additionally, it is interesting to note that Wongsuphasawat et al. only used students in their study, who are hardly experts on the subject. This leads us to believe that experts would likely have found the requested patterns and anomalies not only quicker but also with more precision.

Eick et al. [4] also tested their own software to compare their findings with those of experts, who manually inspect the same log files. They note that the sheer size of the used logs is a bottleneck for the analysis process. They claim that this is due to the fact that the files need to be analyzed multiple times to verify time related relationships. They themselves tested their software, and competed against experts using traditional methods. Not only did the authors find all patterns, which the experts reported in 20% of the time, but they also found more patterns because they could delve deeper into the analysis.

The user studies conducted by both Wongsuphasawat et al. [3] and Eick et al. [4] show that visualization techniques can help with the analysis of log files. But it remains to be shown where to place the project in terms of what ASML has already done regarding the presentation and analysis of event logs.

The rest of this chapter is structured as follows. First Section 2.1 describes the general structure of the provided data. It is followed by Section 2.2 which discusses related ASML as well as academic work. This chapter is concluded with a short description of the requirements of the prototype.

2.1 Data

Logs were used as early as the 1700s, take for instance a ship’s log. This log was used to track the cargo as the ship travelled between ports, as well as the distance travelled within an amount of time. Nowadays, the concept of a log has spread to a variety of other applications.

Any process can generate an event log. In general, this produces a file on a system. Events are generated to mark the progress or whenever it encounters an error. Each logged event is time stamped and appended to the event log. So over time, the content of a log grows. Hence, one can view an event log as a chronologically ordered stream of time stamped events.

Often, the events that are logged are structured. This makes it easier for automated tools to extract the abundance of information in the event itself. The exact structure of the information is dependent not only on the source of the event log, but also on the type of the event. For instance, consider a wafer scanner that is producing wafers. When a wafer was exposed, it is important to log whether this was done successfully or not. Such an event would have this additional information somewhere in its corresponding log record.
The structure of a log record is also dependent on the source of the event log. Ideally, each log record is formatted consistently throughout the event log, enabling a parser to extract generic information for each logged event. Typically, the severity of an event, for instance, is included in the initial few characters of a log record. This severity indicates how important the event was. It could, for instance, only be a notification on the progress of a task, while in another case, it might indicate that the value of a sensor exceeded its threshold.

The variety of event logs makes it harder to generalize them, hence it is difficult to define techniques that can be applied to many different event logs. As a result, techniques are often either very specific for a certain type of log or very general (ignoring a lot of information). An attempt at formalizing the structure of a general event is made in the next paragraphs.

There are some (software) processes that log events. These processes create a stream of log records. These records are formatted consistently while still allowing for some freeform text. This last condition is important to allow extra information per event (type). Let’s say that some event $e_i$ occurs which needs to be logged. A corresponding timestamp $t_i$ and message $m_i$ are generated and written to the event log. The problem with these records is that it is basically freeform text, hopefully with some constraints on the format that stem from both the source of the event log as well as the type of the event (as described above).

Knowledge of this formatting can be used to extract a finite list of two or more properties. There are at least two, because we always know the time at which an event occurred as well as the textual representation of the event in the event log. A property is basically a mapping from the property identifier $p_k$ to its value $v_k$. This mapping can be as complicated or simple as one can imagine. The most trivial mapping imaginable would be to map a given event to its time and text in the log. See also Figure 2.1.

![Figure 2.1: Generic generation of events and the structure of an event log. Whenever an event $e_i$ is generated, a formatting is used to generate both a corresponding timestamp $t_i$ as well as a textual representation of the event $m_i$. This time stamped textual representation is appended to the event log. Whenever information about this event is needed, knowledge of the formatting as well as the specific information about the event type can be used to extract useful information from the record. In the box to the right, the trivial properties (the time and the record) of an event are shown.](imaginary-url)
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Notation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>$i$ numeric</td>
<td>The identifier (numeric) of the event. Note that this identifier is not logged explicitly. It is used by the log parser to be able to refer to the event.</td>
<td>5729</td>
</tr>
<tr>
<td>Timestamp</td>
<td>$t_i$ mm/dd/yyyy hh:mm:ss</td>
<td>The point in time at which the event occurred.</td>
<td>11/06/2014 20:32:08</td>
</tr>
<tr>
<td>Message</td>
<td>$m_i$ String</td>
<td>The textual representation of the event.</td>
<td>“Csv file . . .”</td>
</tr>
<tr>
<td>Severity</td>
<td>$sev_i$ String</td>
<td>The severity of the event. This is one of the following: event, warning, error or alarm.</td>
<td>Event; Warning; Error or Alarm</td>
</tr>
<tr>
<td>Error code</td>
<td>$code_i$ CC-HHHH</td>
<td>The error code of the event. Ideally this should be unique for each type of event. Unlike the name suggests, this code exists for all kinds of events rather than just the errors</td>
<td>ER-0FFF</td>
</tr>
<tr>
<td>Component</td>
<td>$comp_i$ String</td>
<td>The software component that generated and/or logged the event. This property can be derived from the first two characters of the error code.</td>
<td>Error Handler</td>
</tr>
</tbody>
</table>

Table 2.1: Structural event attributes. Shows some attributes and their notation for an event $e_i$.

The additional fields that can be parsed from an event $e_i$ in ASML’s log files are listed in Table 2.1. Note that this is not an exhaustive list. Additional attributes can be parsed from events but these are not explicitly used in this project.

Typically, an event in a log represents but a portion of a task. Again, imagine the wafer scanner. Wafers are in general not processed one by one, since switching the optical mask, the so-called reticle, would simply take too long. Hence wafers with the same reticle are batched into lots. The event which marks that a wafer has entered/exited the wafer scanner, does not describe the whole lot. However, the group of events pertaining to those wafers as well as the lot based events do describe the lot task.

Assuming that some global formatting is used, as described above, we can derive event types from the records in the event log. With knowledge about these event types, we are able to distinguish which events represent such a task. Using this knowledge, we can abstract from the individual events and distill a more abstract notion of the task.

Events are logged strictly chronologically, so when two tasks are executed in parallel, the events belonging to the first and the second task are interleaved in the log. As the number of parallel tasks increases, following the flow of the process becomes more and more difficult. Knowledge of the types of events can then be used to disambiguate parallel tasks, by grouping the events by corresponding task.

A mapping is made from the events to a set of tasks. ASML distinguishes tasks by knowing
which events mark the start and end of a task. All events that are logged in between these boundary events are assumed to be a part of this task. As a result, events can be a part of multiple tasks. Figure 2.2 shows two parallel tasks $T_j$ and $T_{j+1}$ that are distinguished by the events marking the start and end of a task.

The occurrence of an event can also yield additional information about the associated task. Suppose that a wafer has been exposed. This would be logged with an exposure event. The presence of this event can be used to derive that during the lot production task one wafer has been exposed. The freeform text generated from an event can be used to derive additional information about it. Suppose that a wafer has incorrectly been exposed. This still means that an exposure event is generated. The knowledge of the failure would then be logged in the freeform text. Using this technique more complex properties of the task can be derived.

The attributes for events are rather straightforward, but the extracted attributes for the tasks are a bit more involved. The basic attributes such as the start and end times as well as the type are trivial. However, as the type of a task varies, different kinds of attributes can be derived. Given a lot task for instance, the number of exposed wafers can be derived by counting the number of events marking the exit of an exposed wafer. Other types of tasks might have different kinds of additional attributes. There are a number of such task specific attributes. Only the number of wafers related to a lot is used in the prototype. Hence the attributes for other tasks are simply omitted. Table 2.2 gives an overview of the attributes that are associated to a task $t_j$. These attributes are extracted by an ASML software library.

To get an idea of the magnitude of a typical event log generated by a wafer scanner, several metrics were calculated using three different machines. The results thereof are shown in Table 2.3. Typically, one of ASML’s event logs describes the operations of a single machine over the course of a day. However, when this log becomes too large it is split into multiple files. As an illustration, an excerpt event log is shown in Table 2.4. Note that for confidentiality reasons, parts of the events are either omitted or changed. As shown in Table 2.2, each of the tasks has a type of which ASML distinguishes 155 different kinds of which about 60 are used often. Appendix A gives an overview of these types including how often per day they occur on average.

From Table 2.3 we can see that there is a huge amount data being logged each day. It shows that it takes a little over a second for a wafer scanner to encounter an event that has to be logged. Despite that this frequency is huge, these logs do not even contain all occurred events. ASML filters the logged events such that only once per wafer the progress is logged. As a result mostly
Table 2.2: Structural task attributes. Shows attributes and their notation for a task $T_j$. Note that a part of the text example has been blurred for confidentiality reasons.

bad events are logged at almost 1Hz.

To complicate things a little more, ASML introduced the concept of “Main Events”. These are events that are considered to be important enough to be shown at the same level as tasks. An example of such an event could be the loading of a new reticle in the machine, the so-called “pod placed” main event.

Table 2.3: Several metrics of an average event log. These were calculated over the course of a half year and averaged per day.
<table>
<thead>
<tr>
<th>Error code</th>
<th>Timestamp</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>KD-0001</td>
<td>05/11/2013 00:04:04</td>
<td>ERROR</td>
<td>Failed to store periodic Color-diff data.</td>
</tr>
<tr>
<td>IH-1527</td>
<td>05/11/2013 00:04:07</td>
<td>EVENT</td>
<td>Prod cex succeeded.</td>
</tr>
<tr>
<td>ER-0FFF</td>
<td>05/11/2013 00:04:18</td>
<td>EVENT</td>
<td>DEACTIVATE: OK.</td>
</tr>
<tr>
<td>IH-1527</td>
<td>05/11/2013 00:04:18</td>
<td>EVENT</td>
<td>DEACTIVATE: OK.</td>
</tr>
<tr>
<td>IH-1527</td>
<td>05/11/2013 00:04:18</td>
<td>EVENT</td>
<td>DEACTIVATE: OK.</td>
</tr>
<tr>
<td>PG-48B4</td>
<td>05/11/2013 00:04:18</td>
<td>ERROR</td>
<td>Queue request not accepted (state: PWKXQUEUE-REJECTED-WAIT-LIMIT).</td>
</tr>
<tr>
<td>PG-48B4</td>
<td>05/11/2013 00:04:18</td>
<td>ERROR</td>
<td>Queue optimum adjustment failed.</td>
</tr>
<tr>
<td>PG-48B4</td>
<td>05/11/2013 00:04:18</td>
<td>ERROR</td>
<td>Queue optimum adjustment failed.</td>
</tr>
<tr>
<td>IH-1527</td>
<td>05/11/2013 00:04:35</td>
<td>EVENT</td>
<td>LO-8138.</td>
</tr>
<tr>
<td>IH-1527</td>
<td>05/11/2013 00:04:37</td>
<td>EVENT</td>
<td>IH-1527.</td>
</tr>
<tr>
<td>IH-1527</td>
<td>05/11/2013 00:04:37</td>
<td>EVENT</td>
<td>KD-0002.</td>
</tr>
<tr>
<td>IH-1527</td>
<td>05/11/2013 00:04:37</td>
<td>EVENT</td>
<td>IH-1527.</td>
</tr>
<tr>
<td>IH-1527</td>
<td>05/11/2013 00:04:37</td>
<td>EVENT</td>
<td>IH-1527.</td>
</tr>
</tbody>
</table>

Table 2.4: An example of a part of a log file. Note that some parts have been omitted or replaced for confidentiality reasons.
Chapter 2. Problem analysis

•

→

2.2 Related work

Visual data exploration aims at integrating the human in the data exploration process, applying its perceptual abilities to the large data sets available in today’s computer systems. The basic idea of visual data exploration is to present the data in some visual form, allowing the human to get insight into the data, draw conclusions, and directly interact with the data. This section discusses some visualization techniques that enable users to visually explore log data.

The following sections describe several techniques used to visualize event logs. These are categorized according to the following types: Glyph based (Section 2.2.1); Non-glyph based (Section 2.2.2). Finally a short overview of tools that ASML uses is given in Section 2.2.3.

2.2.1 Glyph visualizations

Remember that an event log is nothing more than a list of time stamped representations of events. One way to visualize this sort of data is to simply take a time axis and draw all events on that time axis. However, it is not exactly clear how to draw events. One can use both text as well as shapes to indicate that an event has occurred at a specific time. These graphical representations are called glyphs. The actual shape, color, size, location, and orientation can be used to convey additional properties of the displayed events. A simple example of such a visualization is shown in Figure 2.3. Here events are placed on a time scale at when they occur.

Eick et al. [4] propose an application (SeeLog) that basically displays a zoomed-out version of an event log to create the visual impression of a photo-reduced event log. They use a granulated ordinal time axis, split into one-hour columns. Within such a column a record is displayed as it would be shown in the event log: a sequence of lines. They claim that experts can identify events by the length and shape of the record in the event log.

Additionally, they derive several attributes from each record in the log, from which one is chosen to drive the visualization. This attribute is used to color the individual records. See also Figure 2.4.

Ma and Hellerstein [5] take a similar approach. They also parse additional attributes from records in a log. They propose to use a basic scatterplot to discern patterns in the data. Figure 2.5 shows a scatterplot of logs produced by a server network. They plot the hosts against the time, so one can see which host logged an event at what time. Immediately it is apparent that two hosts are misbehaving as they continuously log records (Pattern 1). Patterns 2-4 are repeated several times, which may indicate recurring scheduled behavior or a persistent problem. Upon closer inspection, pattern 4 shows several, simultaneous authentication failures across a number of hosts. It seems that this is an attempt at a security intrusion.
Figure 2.4: A single hour column of log records (rotated ninety degrees). The color indicates the event types, while their shape can be seen as a photo-reduced version of the event record. This reduction allows experts on the event log to quickly identify the event that they are looking for.

*Source: Eick et al. 1994 [4]*

Furthermore, Ma and Hellerstein propose two additional views. One displays the actual record as can be found in the event log, while the other displays aggregated information on the displayed events. It is also good to note the integration of interaction. The scatter plot allows for zooming and filtering, when this is done, the additional views are updated accordingly.

All above-described visualization techniques use an absolute, continuous time scale. But in some cases, the absolute time might not be overly interesting. For instance envision a hospital. In this hospital there are many patients all with their own files. Each of these files can be seen as event logs. When there is such abundance in different logs, one might want to be able to compare two or more of these logs. Since it is highly unlikely that these patients exhibit the

Figure 2.5: Events in a server network. The time is plotted (horizontally) against the host (vertically). Pattern 1 shows continuously misbehaving hosts. Patterns 2-4 are repeated several times. Patterns 2, 4 and 5 show a sequence of rapidly succeeding events.

*Source: Ma and Hellerstein 1999 [5]*
same symptoms at roughly the same time, one might want to shift each individual log to enable users to compare them to provide a common context.

This is exactly why Wang et al. [6] developed Lifelines 2. This enables users to quickly examine and compare a number of event logs. To do this, all event logs can first be filtered on the occurrence of one or more events. All remaining logs are stacked vertically.

To reduce the effort needed, Wang et al. [6] propose to align each record based on a specific event in each log. They call these events sentinel events. Figure 2.6 shows that what the logs would look like when they are aligned by the first Pneumonia and Influenza related complaint. Here we can easily see that the first three patients had complaints diagnosed as Asthma, within a month prior to their first Pneumonia and Influenza related complaint. The last two patients did not have such an event.

![Figure 2.6: Patients with Asthma and Pneumonia complaints. All traces are aligned on the first Pneumonia and Influenza related complaint. Source: Wang et al. 2008 [6]](image)

2.2.2 Non-glyph visualizations

Glyph visualizations can be useful to place events into temporal context. The whole idea of a visualization is to provide a bigger picture. But as events are generated at increasingly higher rates, glyph visualizations have to zoom in significantly to enable users to discern the individual glyphs. This obviously breaks the added value of glyph visualizations. Hence it makes sense to investigate other techniques.

In some instances, people are actively monitoring systems as they are running. For instance, ASML’s engineers often use prototype wafer scanners to test their software. Acquiring time on these machines is difficult since there are relatively many engineers that need to use few
machines. To optimize their usage, engineers often monitor these tests by manually inspecting the corresponding event log as the test is being executed. This is done using a console like application in which records are being appended to the bottom of the display, causing all previous records to scroll towards the top of the display.

Typically, these machines log records at a rate of approximately one record per second. This gives the engineers a second (on average) to read a whole line. Obviously, this is a bit difficult. For this type of users, Mansmann et al. [7] developed an application that gives users more time to view records. They propose an application called Streamsqueeze, which positions the log records such that users have more time to read them, but still allow users to trace the records easily.

StreamSqueeze assigns more screen space to the latest events, thus raising interest in more recent events. It allows for better readability by keeping records fixed on the display for a longer time than traditional methods. Finally, it enables traceability through predictable animations. It displays records as colored blocks with text that are placed in a number of columns. The color of each block is based on the type of event.

As soon as a new record is generated, the oldest record of the first column is replaced by the new record. The replaced one is then recursively placed in the next column. Figure 2.7 shows a small example of this update scheme. Here, two records are consecutively inserted. The records are represented by their index in the incoming stream. When the fifteenth record is generated, it replaces the previous (fourteenth) record. This record in turn replaces the twelfth record in the next column (indicated by the red arrows). The same technique is then applied to the sixteenth record (indicated by the grey arrows). This scheme ensures that half of the time items are static, thus improving readability of the records. Furthermore, it improves the traceability of events by making sure that they only move to adjacent positions. For a more concrete example, see Figure 2.8.

On the other hand, the records are interleaved in the columns. So in the same column, one can no longer see which event occurred before the other.

Wongsuphasawat et al. [3] continued the work of Wang et al. [6] using icicle plots. They state that traditional event log visualization tools cannot answer questions requiring an overview
Chapter 2. Problem analysis

Figure 2.8: Example of StreamSqueeze showing news items as they arrive.


of the data. E.g., a question such as “What are the most common transfer patterns between services within the hospital?” requires examination of all records one by one. Being unable to see all records on the screen at once makes it difficult to spot any pattern.

They describe the LifeFlow application that groups records based on specific sequences of events, and displays aggregated information. In case more details are needed, users can select a specific set of records, which are then shown in a view similar to Lifelines 2.

To be able to construct an icicle plot, a tree of events needs to be constructed. A path in such a tree corresponds to one or more records of the original set. Whenever two records are contained in the same node, this means that they have the same event prefix.

For a small example, see Figure 2.9. In this case four traces are considered. Each of them starts with the same blue event. Hence, they are grouped together into a blue tree node. Each trace also has a pink event directly following the blue event, hence the pink node. After this, top two records continue with red events while the bottom two continue with green events. Hence the tree is split into a red and a green node. This method can then be continued until all records have been processed completely. This structure is visualized using an icicle plot. The width
of a node corresponds to the number of logs that have that node in common while the distance between two nodes is an indication of the time between the events represented by that node.

During this process, a lot of information is lost even though it can be valuable. For instance the time between two events is in general important. To make use of this information it is aggregated per node in the tree. This way, one can easily estimate and display a distribution of specific properties of each possible event trace.

Now, let’s inspect a real life example in detail. In this case, events consist of patients being transferred from one place to the next and leaving within a hospital setting. In Figure 2.10, we can immediately see that the majority of patients that arrive at the emergency room, are immediately discharged and leave the hospital. I.e., it is relatively easy for users to get an intuition of distribution of sequences, enabling them to analyze frequent and infrequent behavior. The sequence highlighted in Figure 2.10 is a good example of a “peculiar” sequence. Here, after dying two patients are moved to the ICU. This can be defended by saying that the patients might have been resuscitated. But, considering that the movement to the ICU occurred on average thirteen days post dying, a myriad of zombie-apocalypse movies spring to mind. Most likely, this sequence shows an inconsistency in the data.

2.2.3 ASML

Not only the academic community but also ASML has made tools that provide insights into Event logs. This section discusses several of these tools. For confidentiality reasons, no screenshots of these tools have been provided.

First, the Event Log Viewer (ELV) displays the events logs as is. It is a console window on a wafer scanner itself. The ELV appends the event records as they occur to the bottom of the window. As discussed in Section 2.2.2 this is not ideal since whenever an event occurs all records are shifted upwards, making it difficult to trace read records. To counter this, Stream-squeeze (Mansmann et al. [7]) was developed. The disadvantage of this application however, is
that it becomes harder to identify which event occurred before another.

A PC-based application that ASML uses to diagnose wafer scanners is called the System Diagnostics Tool (SDT). This tool is used not only to diagnose problems automatically, but also to inspect events and tasks in a textual manner.

Since SDT is used by a variety of people, the interface is kept as simple as possible. The tasks are displayed in vertical list sorted by their starting times. When needed, details of tasks can be displayed as well as lists of corresponding events. This gives a coarse idea of what the wafer scanner has been doing. Typically, approximately twenty tasks are shown in this overview. Given that a task takes about ten minutes (see Table 2.3), this will show approximately three hours’ worth of tasks.

The downside of the way tasks and events are displayed stems mainly from the fact that the data is presented in a textual manner. When trying to find temporal patterns, text is not exactly ideal because it uses the human cognition system to analyze the text rather than the concepts that are described by the text. The visual system of the human brain is highly trained to find (ir)regularities in images, by exploiting this, it is made easier to understand data and find deviations therein. Furthermore, three hours’ worth of tasks hardly provides a long-term overview of what a machine has been doing.

Moreover, since the tasks are displayed as a list it is hard to identify deviations from temporal patterns. Not only the distance between two tasks has been hidden in textual information, but also the end time, making it more difficult to detect and analyze parallel behavior. Note that SDT can do much more than what is detailed here. For instance, it is also able to display statistics about lots as and plot trends in the scanner parameters.

2.2.4 Discussion of related work

A number of different techniques have been discussed that have been developed either in the academic community or by ASML itself. What remains to be seen is how well the academically developed methods are applicable to the event logs as provided by ASML. To be fully able to analyze these methods it is important to discuss the differences in the data as well as application of the methods.

The data used to illustrate the methods described in Sections 2.2.1 and 2.2.2 are different for each of the techniques. To be able to compare these data, they are classified according to the aspects listed below.

Data range:
The typical range (hours/days/months) at which the data are inspected (Section 2.2.4).

Generation frequency:
The typical frequency at which new events are generated (Section 2.2.4).

Severity:
The typical distribution of severity of events (Figure 2.11b).

Diversity:
The typical diversity of the events being generated (Figure 2.11b).

Regularity:
The typical regularity of the interval at which new events are generated (Figure 2.11c).
Traces:

The typical amount of traces that are present in an event log (Figure 2.11c).

Let’s start with the data used by Eick et al. [4]. Similar to ASML they visualize the log of a single machine, in their case a printer. Unlike ASML’s logs, these are not regulated, and hence they are extremely cluttered with unimportant records: in about half a day 30,000 lines are appended to the file. A quick calculation yields that lines are appended to the log at about 0.5Hz. This is comparable to ASML’s NXT systems. Unlike with ASML logs, an event can be split over multiple lines. So events are being generated slightly less frequent. Furthermore, Eick et al. claim that within a typical hour about 200 different types events occur. From Table 2.3 we know that this is similar to the different types of events generated in an hour by one of ASML’s wafer scanners.

The data used by Ma and Hellerstein [5] describes the workings of a number of computers in a network. According to Ma and Hellerstein about 3,000 events are generated with about 20 different types over about 160 different computers. Of these events, most indicate that a problem has occurred. However, like with the printer logs of Eick et al., this is not regulated.

Wang et al. [6] and Wongsuphasawat et al. [3] use the same type of data. They use the records of a hundred of patients in which complaints as well as examinations are used as events. Assuming that the doctors of the hospital perform their work well, it is realistic to assume that there are months, if not years, between two events of a single patient. On the other hand, if a patient is admitted to the ICU it is likely that events occur in rapid succession.

Finally, Mansmann et al. [7] use news items from some 4,000 different sources, which generate approximately 100,000 different news items per day. So per source approximately 25 items arrive per day. Note that as this entails new records, all events are unique.

The discussion of the data is summarized in Figure 2.11. Note that Wang et al. [6], Wongsuphasawat et al. [3] and Mansmann et al. [7] make no distinction between good or bad events. So in Figure 2.11b these are located on the axis for the diversity.

Now that the data has been categorized according to some aspects, it is good to identify the types of actions that the tools were designed for.

Eick et al. have made an attempt at showing an event log in a single screen enabling users to quickly identify events that they are looking for using the shape of the text. This would not have the same effect on ASML’s log files, since here all events are placed on a single line. Furthermore, this ignores a lot of the structure that has cost ASML years to incorporate in their logs. Also, this would only allow experts on the event logs to find the events that they are looking for, while at ASML there is not a single person who knows everything about their machines.

The main usage of EventBrowser by Ma and Hellerstein [5] is to enable users to identify anomalies. They do so by enabling users to identify frequently occurring computers and events visually. While this does look promising, the sheer scale of ASML’s event logs is simply too large to be displayed in one go.

Wang et al. [6] and Wongsuphasawat et al. [3] use their data to be able to compare different traces of a patient’s medical record. While this technique would be useful for ASML, the scale of the used traces is far smaller than that of a single Lot task, let alone the event log of a whole day.

Streamsqueeze was designed to enable users to have more time to read event records in real
Chapter 2. Problem analysis

(a) Data range and generation frequency.

(b) Data severity and diversity.

(c) Data regularity and number of traces.

Figure 2.11: Schematic comparison of the data used in related work.

- ASML
- SeeLog Eick et al. [4]
- EventBrowser Ma and Hellerstein [5]
- Lifelines and Lifeflow Wang et al. [6]
  and Wongsuphasawat et al. [3]
- Streamsqueeze Mansmann et al. [7]

time. While this is fine for uncorrelated events, the temporal relation between events generated by wafer scanners is important. Hence, a streamsqueeze-like application would not be overly useful when inspecting ASML's event logs.

2.3 Missing pieces

The previous sections discussed what both ASML and the academic community have developed to inspect event logs. Yet, these do not satisfy all of ASML's needs.

When such a machine fails, there are a number of tools that either analyze the fault automatically or allow an engineer to do it manually. So there is no lack in support for diagnosis.

However, in general a wafer scanner does not stop working out of the blue. The precursors of this failure would go unnoticed in the existing used tools, since these tools are specialized in quickly solving the issue at hand. When the main problem is solved, it is desirable to get a bigger picture of the machine. This would show what happened prior to the fault. Despite this, ASML lacks a tool that allows them to quickly get an idea what a wafer scanner has done over a longer period of time (a day containing approximately 60,000 events).

Additionally, there is no clear view on what an engineer is doing with a wafer scanner. One of his actions might well be the cause of some issues. When these interactions are logged, it would be a good idea to show them on a larger scale.
2.4 Requirements

Having discussed what the provided data look like and what already has been done, we describe the requirements, which the prototype has to satisfy. These have been gathered from a series of interviews with a number of ASML’s employees, as well as the feedback from Dr. ir. P. van den Hamer and Prof. dr. ir. J.J. van Wijk. A short list (including explanations) of the requirements is given below:

What First, it is important to know what information should be visualized:

W1. **Display how well a wafer scanner has been performing.**
   On a top-level, how well a machine has been doing is one of the key aspects of a wafer scanner. Hence this aspect should be displayed in the visualization software.

W2. **Display what a wafer scanner was doing.**
   Externally, one cannot see what a wafer scanner has been doing. There are some interfaces that provide additional information. However, access to these interfaces is restricted. All that engineers can do is inspect the event log and hope to derive some insights using a console of the scanner. It would help if more insights could be shown.

W3. **Tasks and main events**
   When a user is inspecting a machine, it is desirable to show more detailed information over a specific interval. Since the concept of a task is already used in ASML’s log inspection tools, it makes sense to visualize these. Additionally, since ASML deems the concept of main events to be important enough to show them on the same scale as tasks, these should also be included in this visualization.

W4. **Events**
   Tasks abstract from a huge amount of data. For experts, this data could make sense. Hence it is useful to display the events.

How For the actual visualizations, the following requirements apply:

H1. **Link to a time scale.**
   To be able to (visually) identify temporal patterns, it is useful to provide a time scale. This enables users to extract temporal patterns in the data using the visual cortex rather than having to actively process the data.

H2. **Link to textual context information.**
   Besides temporal relations between tasks and events, there is a lot more information hidden in the textual representation of the events and tasks, other than the time at which they occurred. So when more details are needed, the user should be enabled to request more information.

H3. **Provide an overview of the data.**
   Given that there are hundreds of tasks, the prototype should enable the user to quickly identify what the machine has been doing over a whole day, rather than
limiting them to a couple of minutes. On an average there are hundreds of tasks and
tens of thousands of events (see Section 2.1). It is unlikely that these events can be
shown on a display, let alone that the user’s cognitive system can process them all
at once. Hence it makes sense to provide an overview thereof. A user should be
able to view data of at least twenty-four hours and be provided with an uncluttered
overview.

H4. Provide details when needed.
An overview often abstracts from a lot of data. When desired, the user should be
able to see the details of this abstracted data.

H5. Confidentiality
Chip factories cost billions of dollars and produce similar amounts in revenues. De-
tails of what these factories produce, and how they are run are sensitive information
that is owned by ASML customers. In contrast, details of how the scanner operates
internally is ASML confidential “intellectual property”. In general, customers share
event logs with ASML (for diagnostic and learning purposes) under the condition
that the information is handled as confidential data. Thus the information summa-
rized by this tool is equally sensitive as the source (event log) data. For this reason,
some details such as specific machine identifiers are anonymized.

Usability The population of first line support engineers is very diverse. The only common ground
is that they have had training, in a matter of weeks, on how to diagnose and repair a wafer
scanner using the toolchain ASML provides. Hence, the tool should be easy to use by
a variety of people. There are many guidelines that tell how to achieve this. The most
important ones are listed below:

U1. Responsive.
One of the key issues with most software is performance. If the user has to wait for a
significant amount of time, it is likely that his view on the software will be negative.
Therefore, it is critical that the software should be responsive. If it is inevitable that
the user is waiting on the application, he should be notified of the progress made.
Feedback on user actions should be given within 300 ms.

U2. Uncluttered
A display filled with many data items is hard to process. At some point, there’s too
much information on the screen to be processed at once. As a result, the user would
have to limit himself to only a segment of the data, losing focus of the bigger picture.
To minimize this effect, the prototype should be uncluttered.

U3. Consistent
One would not want to confuse users by representing the same thing or action in
more than one way. Hence, it is important that the same concepts and actions are
represented in the same way.

U4. Familiar
It is important that familiar standards and conventions are used. This will make
grasping the simple things effortless allowing the user to focus on the difficult ques-
tions.

**Context**  Lastly, it is important to know under which conditions the prototype should run.

**C1. Single machine**
As stated in Section 2.3, the prototype is not meant as a means to compare machines. Furthermore, one wafer scanner does not affect the working of another, so there is no real need to compare multiple machines along a synchronized time axis. Hence, the prototype should be able to visualize the history of a single wafer scanner.

**C2. Machine types**
ASML produces different types of wafer scanners. It is sufficient to be able to visualize event logs of AT/XT, NXT and NXE machines.

**C3. Independent**
Usage of CPU time on a wafer scanner is strictly regulated. Furthermore, the prototype is meant to run on historical data, so there is no need for it to run on a scanner.

**C4. Microsoft windows**
Throughout ASML, Microsoft windows is widely used. It makes sense to ensure that the prototype runs on this operating system. Any other operating system is optional but not required.
Chapter 3

Log browser

As stated in Chapter 2, it is useful to visualize event logs. Methods from the visualization community seem to be inappropriate for ASML’s logs or needs (Section 2.2.4). ASML’s log viewing tools however, do not exploit the visual capabilities of the human brain (Section 2.2.3). This allows us to develop a prototype that is keyed towards ASML’s data and needs. Section 3.1

Figure 3.1: The four different views are indicated by the curly braces at the right. Following Shneiderman’s mantra, an overview of the data is provided in the top two views that show the data on the same scale. When desired, the user can zoom in on an interval such that the prototype shows the tasks. When even more details are needed, the user can zoom in further such that the events are shown at the bottom. This zooming is indicated by the red rectangles and lines. The dashed rectangle at the left shows the legends and additional statistics for each of the views. Note that the red lines are only annotations, these are not part of the prototype.
gives an introduction to the developed prototype. This is followed by more detailed descriptions of the individual components in Sections 3.2 to 3.5.

### 3.1 Overview

Shneiderman [8] defined the widely used information seeking mantra:

*Overview first, zoom and filter then details on demand.*

This mantra has been leading in the development of the prototype. First an overview of the data is given by the productivity and activity views. When more information is needed, the user can zoom in on the data. Finally when a user finds some interesting nugget of data, a tooltip can show additional details on demand.

Requirements W1 to W4 indicate which aspects of the data have to be visualized and why. These different aspects do not lend themselves well to be shown on the same scale, so it does not make sense to show all of these aspects in the same view. To keep things in perspective, all different aspects remain on the display. This zooming is shown in Figure 3.1.

Providing an overview is an integral part of the goal of this project. To do this, an overview of how well (Requirement W1) and what (Requirement W2) a wafer scanner has been doing should be provided. These overviews are described in Sections 3.2 and 3.3 respectively. From Shneiderman’s mantra, we know that this should show an overview of all of the data. So this would typically show data at the scale of about 24 hours or more.

The detailed views should not attempt to visualize all the data at once, as this would result in a huge amount of visual clutter. The view showing the tasks has been designed to show at most a couple of hours’ worth of tasks. Given that events are generated at almost 1Hz, the maximal data range for the view showing the events is limited to tens of minutes.

Figure 3.2 shows a wafer scanner that experiences some intervals with low productivity. While most of these are caused by maintenance being done, one of these intervals actually displays a problem. This problem was quickly solved within an hour.

### 3.2 Productivity view

The top-level in which a client is interested, is the productivity (Requirement W1). When a client’s wafer scanner does not produce the promised amount of wafers, he is not satisfied. Furthermore, the productivity of a machine gives a good indication of how well a machine has been performing.

From Section 2.1 we know that for some tasks there are some additional properties that are derived from the corresponding events. For a lot task $T_j$ this means that the amount of wafers is given as a triple:

$$ (e_j, r_j, u_j) \quad \text{ where } \quad e_j \quad \text{represents the number of exposed wafers} $$

$$ r_j \quad \text{represents the number of rejected wafers} $$

$$ u_j \quad \text{represents the number of wafers left in the machine} $$
Figure 3.2: This wafer scanner experienced three intervals with significant lower production (A\(_1\), A\(_2\) and A\(_3\)). The first and the last interval experienced lower productivity because the machine was doing a lot of maintenance (the purple in B\(_1\) and B\(_3\)). During the second interval something was wrong with the machine since wafers were rejected (red bars in A\(_2\)). Apparently this problem could be solved by stabilizing the fluid underneath the lens (blue bar in B\(_2\)). In the Task View, part of this second interval is shown. The events of the Lot task indicated by C are shown in the Event View. Here, there is a clear pattern in the generation of events. Since this pattern is repeated twenty-five times, it is highly likely that this sequence of events is repeated for each wafer.

Knowing this, the average productivity over all lots is computed as:

\[
\text{avg prod}_j = \frac{\sum e_j}{\Delta t}
\]

where \(\Delta t\) is the length of the period considered. Note that only lot tasks are considered here.

Computing the productivity of a lot task \(T_j\) seems trivial by dividing the number of produced wafers \(e_j\) by the duration of the lot \(\Delta t_j = \text{end}_j - \text{start}_j\). However, a wafer scanner is a pipelined machine. This means that while a wafer is being exposed, another wafer is being prepared for exposure and others are placed into a buffer. As a result of this pipelining, lot tasks overlap while in fact only one lot is in production at the same time. Wafers from the other lot are only being buffered and prepared for exposure. Hence, the duration of the shown data is less than the sum of the durations of the individual lots (\(\Delta t \leq \sum \Delta t_j\)).

If the productivity were computed in the way described above, the calculated productivity of an isolated lot would be too small. This would become evident when the productivity is compared to the average productivity over a whole day. The effect of the overlap is shown
Chapter 3. Log browser

26 Chapter 3. Log browser

(a) The average productivity is significantly higher than the average height of the bars.

(b) The average productivity equals the average height of the bars.

Figure 3.3: Effects of overlapping lots on the average productivity. The average productivity over the day is indicated by the red lines.

in Figure 3.3a. Here, the average productivity over the whole day (149 wafers per hour) is significantly higher than the average of productivities per lot (131 wafers per hour).

To solve this, the productivity of the lots has to be calculated such that such that the overlap between lots is ignored. To remove the overlap for two lots $T_j$ and $T_{j+1}$, we have the following three options (primed variables indicate the values after the change):

- **Cut off the first lot:**  $end'_j = start_{j+1}$
- **Split the overlap evenly:**  $end'_j = start'_{j+1} = end_j - \frac{end_j - start_{j+1}}{2}$
- **Cut off the second lot:**  $start'_{j+1} = end_j$

These options are shown in Figure 3.4 by the red, green and blue line respectively.

When inspecting a log, it becomes apparent that it is likely that the production of a second lot starts before the first lot task has been completed (see Figure 3.5). Hence, the second option is chosen in which the overlap is equally divided between two lot tasks. With this correction, the discrepancy between the lot productivities and the average productivity is removed (Figure 3.3b).

Given the different types of wafers, the productivity of the lots in a specific period can be

![Diagram of two overlapping lots $T_j$ and $T_{j+1}$ with overlap $\Delta t$, $\Delta t_j$, $\Delta t_{j+1}$](image)

Figure 3.4: Overlap of two lots $T_j$ and $T_{j+1}$. The greyed area shows the overlap of the two lot tasks. Note that $\Delta t \neq \Delta t_j + \Delta t_{j+1}$. The red, green and blue lines indicate how the overlap can be divided over the two lots to make sure that there is no more overlap when calculating the productivity.
Figure 3.5: Boundaries and production of two lots. The purple circles show the events marking the boundaries of the lots, and the blue circles show when a wafer (belonging to that lot) has exited the machine. Here we can see that end of the first lot task occurs well after the last wafer has exited the system. The red line shows the average time between two wafers leaving the system. Note that this screenshot of the prototype has been edited to highlight this point.

Figure 3.6: Schematic visualization of the productivity using a line and a bar chart.

seen as a list of tuples:

\[(e_1, r_1, u_1, start_1, end_1), (e_2, r_2, u_2, start_2, end_2), \ldots, (e_j, r_j, u_j, start_j, end_j)\]

To visualize this data, two standard visualizations come to mind: a line chart with three different lines and a stacked bar chart with different colored bars representing \(e_1, e_2, \ldots, e_j, r_1, r_2, \ldots, r_j\) and \(u_1, u_2, \ldots, u_j\) respectively. For a schematic example showing only exposed wafers see Figure 3.6.

This last option was chosen for two reasons. First, with a line chart is that it not clear where the vertices of the lines should be positioned on the time axis. This could be at the start/end of the lot or anywhere in between. Second, when a bar chart is used, the area occupied by the bars should equal the number wafers produced. This is because the vertical axis measures wafers per time unit and the horizontal axis measure the time unit \((w/h \times h = w)\). Figure 3.7 shows an example where the three different kinds of wafers are shown.

Figure 3.7: Productivity of a wafer scanner over eight hours. This shows regularly exposed wafers (A), remaining wafers (B) and rejected wafers (C). The yellow markers (D) at the top indicate periods with low productivity.
Depending on the reticle, or recipe, a wafer scanner needs more time to pattern the desired image onto a wafer. When enough wafers are exposed, this extra time will significantly affect the productivity of a lot. Fortunately, the used reticles are stored in the lot task as well, and hence it can be displayed. But viewing a large number of lots showing this information can be overwhelming as seen in Figure 3.8.

Again, Shneiderman’s mantra seems to be useful. Too much data is available to be clearly shown on a single display. Zooming in on this data would allow the user to view the information in a clearer setting. This information is hidden when it would clutter the display, when zooming in, more and more details are shown albeit of less data, an approach also known as semantic zooming [9].

The same idea is used with the overlap of lots. When the user is looking at too many lots, there are too few pixels to indicate the overlap between them. When this number becomes too small, the overlap is hidden from the user and hence providing a calmer display.

So three different levels of details are used in the Productivity View. At the highest level (least details) only the productivity of all lots is displayed. One level deeper, changes in the used reticle(s) is shown. At the deepest level, the overlap between the lots is displayed. These levels of detail are shown in Figures 3.9a to 3.9c respectively. Note that only the transition of one reticle to the next is displayed, because the name of a reticle is sensitive information (Requirement H5).

The above-described level of detail is managed automatically. This is achieved by using the average size of a lot on the screen to determine which level of detail is most appropriate. An
example of this management is shown in Figure 3.10.

Finally, to aid the user, periods of low productivity are indicated by the yellow markers at the top of the productivity view. Note that other types of markers could be added, for instance to mark the rejection of wafers or hint at failed tasks. The yellow markers are shown explicitly to simplify the detection of periods with suboptimal performance.

3.3 Activity view

It is useful to have a global overview of what a wafer scanner has been doing. With the Productivity View one can get an idea of how well a machine has been performing. Dips in this productivity, however, are not explained. So it has to be shown what a machine was doing during an interval (Requirement W2).

One would like an overview of what a machine has been doing over at least one day. The tasks describe fairly well what a machine has been doing on a small scale. Showing all tasks over the course of a whole day would result in a cluttered display. To solve this, tasks can be aggregated into bins, where each bin indicates how much time a machine has spent on the different types of tasks during an interval.

A task has been modeled as an activity spanning some interval with additional attributes. To aggregate these, one can sum the durations of the tasks of the same type. This is done by calculating the overlap of each task with some interval and summing this per type.
Figure 3.11: Aggregation of three tasks on an interval. Tasks \( T_j \) and \( T_{j+2} \) take 3 time units while task \( T_{j+1} \) takes 5 time units. Task \( T_j \) is only partially (two-thirds) contained in the interval \([1, 9]\). So in the interval, five \((2 + 3)\) time units are spent on gray tasks and five time units are spent on white tasks. Note that in total ten time units are used while the interval only takes eight time units. This discrepancy is caused by the overlap of task \( T_{j+1} \) with the other two intervals.

The amount of time spent on a type of tasks (type) during the interval \([start, end]\) is calculated as:

\[
\sum_{j \land \text{type}_j = \text{type}} \Delta t_j - \max(start - start_j, 0) - \max(end_j - end, 0)
\]

So for an arbitrary interval, we end up with how much time has been spent on some type of task. For a small example see Figure 3.11.

Computing this sum over all intervals can take a long time when a brute force approach is used \(O(n \cdot J)\) where \(n\) and \(J\) represent the number of intervals and tasks respectively. To speed this process up, the set of intervals is stored in an interval tree [10]. Using this interval tree, all intervals with a task can be found in \(O(m + \log n)\) time where \(n\) and \(m\) represent the number of stored and returned intervals respectively.

When tasks overlap frequently, the summed overlaps can be greater than the duration of the interval. Counter intuitively, this means that a wafer scanner can spend an hour and twenty minutes on lot productivity in an interval of only an hour. Figures 3.13a and 3.13b shows how this would look visually. To keep things simple, this discrepancy is kept hidden from the user since this does not add any more information. This is done by scaling the values visualized to 100%. For an example thereof see Figures 3.13c and 3.13d.

While we have described how to calculate the activities within an interval, it has yet to be determined for which intervals the activities have to be computed. One could choose regular intervals, for instance each half hour, or try to find some reference points.

To enable users to relate the productivity to the activities, it makes sense to use the same lot intervals as shown in the Productivity View. To make sure that activities are also displayed when no lots are being produced, the intervals between lots are also used to compute the activities.

In essence, the same kind of data is used as with the productivity view of Section 3.2. Rather than using different kinds of wafers here different types of tasks are used. As stated in Section 2.1, there are a number of events that are deemed important enough to be shown at the same level as a task. A problem with these so-called main events, is that they are completed instantaneously, e.g., the changing of some machine parameters or the removal of a wafer from the machine. As a result, these main events are hidden when using the duration of tasks to create an aggregation. To remedy this, users can easily change a setting to show the number of
occurrences rather than the duration within the reference interval.

Havre et al. [11] propose a visualization that displays thematic variation over time. By changing the width of a “flow” the variation in thematically similar data elements is shown. As such, a ThemeRiver is in essence a “fancy” area chart. Here the values are mirrored along the centerline and interpolated to create a smooth shape. This interpolation is achieved using Bézier splines [12]. Figure 3.12 shows an example of how a ThemeRiver is constructed.

A stacked bar chart seems to be the better choice for the following reasons. First, the people are not used to a ThemeRiver like visualization making it difficult to read and the interpolation is not helping either. Hence the ThemeRiver implementation is conflicting with Requirement U4. Furthermore, the interpolation is not consistent with the productivity view (Section 3.2). Here, the same type of data is visualized un-smoothed. Therefore, the familiar bar chart is selected to visualize activities. For infographics, a ThemeRiver would perhaps be better, because it draws more attention to itself than a bar chart.

In Figures 3.13a and 3.13b it is clear that something odd is happening between two and six o’clock. Upon closer inspection, a lot is aborted (indicated by the red shapes). Here it has taken

Figure 3.12: The construction of a ThemeRiver. This is showing four different task types (A-D) at three different moments in time. The four bars of a stacked bar chart are shown centered on the middle line (dotted). The values are interpolated by a Bézier spline.

Figure 3.13: The activities (unscaled) rendered like ThemeRiver. The activities (unscaled) displayed as a bar chart. The activities (scaled) rendered like ThemeRiver. The activities (scaled) displayed as a bar chart.
Figure 3.14: A schematic example of a Gantt chart. There are four tasks A to D. Tasks B and C can only start after A has been completed. Task D can start when C has been completed. The precedence relation is indicated by the gray lines.

over two hours to reboot a series of drivers, which failed to solve the issue. After a reboot, it took another half an hour before the machine was stable enough to resume lot production.

3.4 Task and Event view

First line support engineers are used to think in terms of tasks and events. In the academic related work (Section 2.2.4), the methods with data of similar scale to ASML’s, all use glyph based visualizations. So it seems to be a good idea to visualize the data using glyphs.

The sheer number of the tasks and events makes it difficult to show them on a single time axis. Therefore, the tasks and events have to be partitioned into different groups. Following the idea of Ma and Hellerstein [5] the different groups are displayed on different lines. Unlike the implementation of Ma and Hellerstein this is not exactly a scatter plot, as the glyphs are always displayed against a time axis (Requirement H1).

Now that it has been decided that glyphs are used to visualize tasks and events, we have to establish how the data is represented visually. The different attributes from Table 2.2 cannot all be shown at the same time by using color to differentiate them, since the contrast between the colors has to be large enough for the human eye to be able to distinguish them. For the tasks, it was decided that the task type should always be indicated by the color of the glyphs.

A task $T_j$ can be considered as an interval $[start_j, end_j]$ with some added information (see Section 2.1, Table 2.2). Intervals are often displayed as bars, where the width indicates the duration of the interval and the positioning along the time axis indicates when the interval started and ended. For instance, look at the Gantt chart in Figure 3.14. The tasks in a Gantt chart have a precedence relation. This means that before a task starts, there may be some other tasks that should be finished first. This precedence relation is not explicit in ASML’s data. However, this does not prevent us from visualizing the tasks in the same way.

Given the number of different tasks in a couple of hours, it is impossible to show them with each task having a unique height. Much like in a scatter plot, the tasks are grouped together to make up for the limited amount of available vertical space. To enable users to focus on properties of tasks that they are interested in, the tasks can be grouped by one of the following properties (see Figure 3.15):

- Task identifier: The identifier of the task.
- Task type: The type of the task.
- Task result: Whether or not the task was completed successfully or not.
These groups are alphabetically sorted. However, when the tasks are only grouped, it is not obvious which attribute is represented by a group. To solve this, a label is drawn for each group, given that there is enough vertical space to draw them legibly.

An example of the Task View is given in Figure 3.16. This shows an interval of four hours where the task type was used for the vertical grouping. In this interval the wafer scanner stops lot production to do a little maintenance. During this maintenance several machine constants are altered.

Note that there are events deemed important enough to be displayed on the level of tasks. The duration of these so-called main events is zero milliseconds, so they are instantaneous. Therefore, bar glyphs do not lend themselves well to display these events. These main events are displayed consistently with the event shown in the Event View. Some examples of such main events are also shown in Figure 3.16.

Not everything can be deduced from tasks since a lot of data is being abstracted. Furthermore, it might be interesting to see which (software) component log an excessive amount of
Chapter 3. Log browser

(a) Grouping by component.

(b) Grouping by error code.

(c) Grouping by Severity.

Figure 3.17: The grouping of events on various attributes. Here the color of the events is determined by the severity attribute.

Events do not have a duration since they are instantaneous. Hence, a bar glyph is not suitable to display events. There is a multitude of shapes that can be used to show different aspects. For an event $e_i$, a specific shape (e.g. circle/star/square/etc.) could be used to indicate the different component $c_i$. To keep things simple, all events are indicated by a circle where the color and the vertical position can then be used to convey information about the attributes of the event.

Again, to enable users to find correlations between different properties of events, the attributes used to determine the color and vertical position can be chosen by the user. The following attributes of an event can be used (see Figure 3.17 for the grouping of events):

- **Component:** The software component that generated and/or logged the event.
- **Error code:** The error code of the event.
- **Severity:** The severity of the event: event, warning, error or alarm.

Note that the groups are sorted and labelled in the same fashion as the groups in the Task View.

Figure 3.18 shows the events of about fifteen minutes of lot tasks. It shows two components which seem to log a many events simultaneously.

Figure 3.18: Events in Event View. Here two components seem to be logging events simultaneously. Note the events with the red border. The events from the “Height & Tilt control expose” are mirrored by an event from the “Error Handler”. Here both the color as well as the vertical position of the events are determined by the component that generated the event.
3.5 Common functionality

According Requirement U3, it is useful to keep many aspects of the prototype consistent. This section lists some of the functionalities that are prevalent in all the four distinctive views (Sections 3.2 to 3.4).

Interaction with time

Requirement H1 states that the data has to be displayed in relation to a time axis. To keep things consistent (Requirement U3), all views use the same interactions with the concept of time. There are three different actions the user can do with the time interval of a view: zooming, panning and selecting. The first two of these actions change the intervals of the view the action is executed on. The third one indicates which data has to be shown in another (more detailed) view.

To formalize these actions, some notation has to be introduced. There are $n$ views $v_1, v_2, \ldots, v_n$. Each of these views shows an interval of time. So for view $v_i$ this is $[s_i, e_i]$. Now the above-described actions are easily formalized (see also Figure 3.19):

Zoom by $\Delta t$: $[s'_i, e'_i] = [s_i - (1 - r) \cdot \Delta t, e_i + r \cdot \Delta t]$

Pan with $\Delta t$: $[s'_i, e'_i] = [s_i + \Delta t, e_i + \Delta t]$

Select $[s, e]$: $[s'_j, e'_j] = [s, e]$ for some $v_j$

Here all primed variables indicate the result variables after the action has been completed. The $\Delta t$ for zooming and panning is determined by the ticks of the timescale. The zoom action zooms in on the position of the cursor. Here $r$ represents the ratio between the space to the left of and to the right of the cursor. A pan action translates the view one tick to the left or right. The interval $[s, e]$ is determined by dragging an area of view $v_i$.

Remember that a selection in view $v_i$ alters the interval of a different view $v_j$. But it is not yet clear which view this is, and we have to provide additional information when an interval is selected, so the data shown in a more detailed view is updated. To this end, each view $v_i$ has a target view $c_i$ which interval is updated upon selection. This changes the selection equation to $[s'_i, e'_i] = [s, e]$ where $[s, e]$ is the selected interval. In Table 3.1 this relation is indicated.

The Productivity View and the Activity View are designed to display data on the same scale up to a couple of days. It makes sense to ensure that their intervals are identical, so the proposition $[s_0, e_0] = [s_1, e_1]$ always holds in the prototype.

Since these views show different intervals, it can be difficult to relate detailed information to what is happening the overview. So a link has to be made from the detailed views (Task and Event View) to the overviews. To enable this, whenever the pointer hovers over a view, the instant in time represented by that location is shown in all other views as a gray line.

![Figure 3.19: A schematic overview of the user actions on time.](image-url)
Table 3.1: The relation between a view and its child view whose interval is updated. Event View does not have a child view and hence selecting an interval in the Event View does not alter anything.

### Legends

As with any visualization, the colors in the prototype have meaning. For the Productivity View, one might be able to deduce what they mean. For the other views, this is not easy. To remedy this, a legend is shown next to each of the views. Selecting one (or more) of the entries in the legend highlights all entries corresponding to the data elements, represented by the legend entry, by drawing a red bar around them. This is shown in Figure 3.18.

One of the downsides of any visualization technique is the lack of accuracy. It is difficult to determine precise numbers based only on a visualization. To counter this problem, some statistics of the shown data are computed and shown in the legend. Figure 3.20 shows an example of these statistics in the legends of all views.

### Responsivity

Requirement U1 states that waiting on the prototype is undesirable. Retrieving and drawing the data can potentially take a long time. To make sure that the waiting time is minimized, several techniques are applied.

Using multiple threads enables the user to interact with the prototype while some data is being retrieved or drawn. This is especially useful when the user is selecting a range of data. One of the threads is then used to retrieve the data real-time, while another is used to show feedback (the selected interval) to the user.
Linkage to textual information
As with any kind of textual information, it is difficult to visualize all information encapsulated in an event log in a single image. To enable users to retrieve this lost information (Requirement H2), hovering the cursor near the visual representation of data items opens a tooltip with the textual information as used in ASML’s log viewing tools. Examples of textual information of the detailed views are shown in Figure 3.21. In a similar fashion, the overviews also show textual details.

Overlapping data
Overlapping data is one of the most persistent issues when visualizing the ASML events logs. This is mainly caused by overlapping tasks and fixed sized glyphs. To allow users to see this
overlap, all glyphs and shapes are drawn transparently. Some small examples are shown in Figure 3.22.

**Hiding views**

A user could be interested in only a selection of the data. For instance, it is likely that he would only be interested in getting an overview of a wafer scanner. In this case, half the screen space would be wasted with the space reserved for the detailed views. To remedy this waste the user has the ability to hide views such that more screen space is allocated to the other views and different tools.
Chapter 4

Discussion

Now that the visualization techniques have been described, we discuss whether the developed views satisfy the requirements of Section 2.4 and allow users to answer the questions of Section 1.1.

Requirements

First, the context requirements (Requirements C1 to C4) are satisfied by construction. Most of these are satisfied by the parsing library provided by ASML.

The what requirements (Requirements W1 to W4) are each satisfied by the view designed for them. Additionally, the Activity View shows how long the scanner has been working on a specific task. This provides information about the tasks that occur during the intervals. Some events are considered to be so important that have been promoted the level of tasks (by the parsing library). This means that these events are also shown in the Task View.

Whether the How requirements (Requirements H1 to H4) are satisfied, is more involved. The linkage to a time scale and the textual information of the data are satisfied by the common functionality of the views. The prototype provides an overview (Requirement H3) and detailed (Requirement H4) information of the data. The sheer scale of the data prevents a view that meets both requirements simultaneously. Hence the two overview and two detail views were developed. Finally, only the Productivity View shows confidential information (Requirement H5) of ASML’s customers. Releasing this information about the used reticles to others is undesired. To prevent this, the Productivity View does not show this information explicitly.

Last, the usability requirements (Requirements U1 to U4) are discussed. Each of the views use similar techniques to speed up the retrieval and the drawing of the data. The Event View cannot do this all the time, due to the scale of the event logs. This problem only arises when events are requested from a file that is not cached. In this case it would take approximately two seconds to retrieve the needed events. This only occurs if multiple files are loaded, which typically is not the case during a single session.

As was to be expected, the overview views are less cluttered than the detailed views. This is due to the smaller amount of data items visualized and the smaller amount of overlap. As for consistency, all views use the same interactions. Furthermore, the colors used throughout the prototype are kept consistent when they indicate similar concepts. For instance, a lot task has the same color as exposed wafers in the Productivity View, similarly the recovery task has the
Table 4.1: Summary of the satisfaction of the requirements from Section 2.4.

<table>
<thead>
<tr>
<th></th>
<th>What</th>
<th>How</th>
<th>Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Productivity</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Activity</strong></td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>Tasks</strong></td>
<td>-</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td><strong>Events</strong></td>
<td>-</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td><strong>Time scale</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Link to text</strong></td>
<td>+</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td><strong>Confidentiality</strong></td>
<td>+</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td><strong>Overview</strong></td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>Details</strong></td>
<td>+</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td><strong>Responsive</strong></td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td><strong>Uncluttered</strong></td>
<td>+</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td><strong>Consistent</strong></td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td><strong>Familiar</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of the satisfaction of the requirements from Section 2.4.

+ Satisfied
0 Somewhat satisfied
- Not satisfied

same color as the rejected wafers. Finally, standard visualization techniques are used to display
the data. Here, the Activity View is an exception, because it is not common to plot time against
duration, which both represent a similar concept.

A summary of this discussion can be found in Table 4.1.

**User feedback**
To analyze the practical use of the prototype, it has been reviewed by two field support engineers,
i.e., a team leader of first line support engineers from Chandler, Arizona and second line support
engineer from Fishkill, New York. They had only a short introduction (approximately half an
hour) into the prototype. Note that a more extensive usability test should still be conducted.

First, both evaluators noted that the prototype quickly provides an overview of the activities
of a wafer scanner:

“[The prototype] provides a nice quick overview of what the [wafer scanner] is
doing and for how long.”

“[The prototype] could help put what the system was doing prior to an error into
perspective. Sometimes you can see a pattern better that way.”

They noted that the tool was useful but it did not suit their need for diagnosis:

“[If the tool is for diagnostics then that is one thing, but it seems like it is more to
get management summary info than to drill down on an issue.”

However, it was not stressed enough that the prototype was not meant as a stand-alone tool.
One or more views could be integrated in existing tools such as SDT:

“I envisaged the tool myself as a prototype (rather than a real tool) and then inte-
grated into SDT or maybe a dashboard (to monitor a bunch of machines).”
As stated here, the prototype was meant to provide a quick overview. More detailed information can be derived from the textual representations of the tasks and events. The prototype can then be used as an index into this data to quickly find interesting periods in time. This would allow users to combine quick overviews using a visualization with all the details available in textual representations of events.

The prototype does provide more context when analyzing a problem and it can be used as a communication tool with management and customers:

“[The team lead] seems to like it to get an overview (as a TL?) and to communicate with the customer. It can also help (if streamlined) to get a quick overview when a system is passed down or escalated?”

The prototype is still not mature and needs more work before it can be exposed to a wider public. Furthermore, they would like to pilot the next versions of the prototype.

“[I]t is just premature and [I] would require a little more guidance on how to best use it”

“Thank you for the time and effort taken to address some of the needs of CS. Look forward to the future pilot of this tool.”

**Similarity to related work**

In retrospect, it always good to see how the prototype relates to other visualizations and whether it can be used to visualize data from different sources.

Requirement U4 states that it is desirable to use familiar visualizations to represent the data. As a result, the Productivity and Activity View are basically bar charts, where the Productivity View has some additional annotations. The Task View shows similarities with a Gantt chart, but there is no precedence relation between the tasks.

From the inspected related work, the Event View resembles EventBrowser by Ma and Hellerstein [5] that is a scatter plot. There are two differences between these visualizations. First, EventBrowser allows users to plot any property of the events on the horizontal axis. By Requirement H1 this is not allowed. Second, other properties of the events can be shown using color in the Event View.

Finally, the prototype was specifically built for ASML’s event logs and the data that can be parsed from them. This makes it difficult to use for other sources of event logs. What is missing from traditional event logs is the concept of tasks. While these tasks could possibly be parsed from other logs, we have not seen any other supplier that does so. But on the other hand, this enabled us to explicitly visualize the concept of tasks, which is a vital concept for ASML.

### 4.1 Scalability

Two of the important usability requirements are that the prototype should always be responsive (Requirement U1) and uncluttered (Requirement U2). Whether these requirements are met,
depends greatly on the data shown. Hence it makes sense to investigate the effects when the data grows either in size or diversity.

**Size**

If the amount of data being generated increases, it would clearly take longer to parse the log. However, since the data is cached in main memory, this would hardly effect the retrieval of the data once the log is parsed. This assumes that there is always enough available main memory to use as cache.

When a typical day is loaded, the prototype requires approximately 72MB of internal memory. If this is increased to incorporate ten days, this amount increases to approximately 81MB. However, note that these amounts of memory were used when the detailed views were showing a relatively small amount of data (six hours and thirty minutes by the Task and Event View respectively). When both detailed views also show all available data, the prototype uses up to 92MB and 370MB for the single and ten day examples respectively. So even in case significantly more data is loaded, it still easily fits in main memory of a modern computer.

Hence, the speed of the visualization is not affected too much when the amount of data increases. But on the other hand, the data of a single file is cached for the Event View. As a result, when events are requested from a different file this has to be reparsed. But on the other hand, this probably does not occur frequently enough to be of too much influence on the user experience. But the effect on the views themselves still has to be analyzed.

The Productivity View suffers least from an increase in data. An increase in the amount of data for this view would mean that more lots are produced in a single day. Still the amount of pixels needed to visualize the average overlap determines what amount of detail is shown. Hence the amount of detail would automatically be adjusted to the new data.

An increase in an amount of lots would increase the number of bars in the Productivity View. This increase in bars could also make the view cluttered. To remedy this, one could introduce another level of detail in which the productivity is expressed over a fixed duration (say half an hour) rather than per lot. This can be seen as using a low-pass filter over the productivity.

Similar to the amount of bars in the Productivity View, the amount of bars in the Activity View would also increase. Introducing the same concept of a low-pass filter would again solve this issue.

Increasing the amount of data shown in the Task View would either decrease the duration of the tasks or increase the amount of overlap in tasks. In both, the maximal interval that can easily be interpreted would decrease. In the first case, the size of the task glyph would decrease making them difficult to identify and interpret. When the overlap is increased, more and more tasks are displayed on the same pixels. This would decrease legibility of the tasks. This last problem also occurs when the number of events increases in Event View.

Concluding, this does not seem to affect the overviews too much. The levels of detail automatically deal with the increase in data. In the worst case another level of detail would have to be introduced. The detailed views on the other hand, suffer more from visual clutter. The easiest way to solve this is simply to zoom in on a smaller interval.

**Diversity**

The diversity of the data does not affect the responsivity of the prototype because the amount of data does not increase. Again whether the views become more cluttered differs per view.
Since there are no other statuses a wafer can have other than exposed, remaining or waiting, it is unrealistic that the diversity of this data would increase. However, given that there are only three different statuses, an increase in the same order would not limit legibility.

The other views behave similarly. Each of them has both issues with the vertical position and size as well as the amount of different colors needed. In the Activity and Task View, the color is determined by the type of the tasks. To be able to distinguish a number of colors, their contrast has to be large enough. In practice, this means that the human eye can easily discern approximately fifteen different colors. From calculations we know that there about sixty different kinds of events that are used. So it is already impossible to show each event type with a unique color, and hence this increase in task type would only mean that more types would be indicated by the same color.

Event View uses different aspects of events to color their glyphs. When the severity of the events is used, there is an ample amount of colors left since, there are only four different kinds of severity. The other aspects of events suffer from the same problems as the tasks.

The vertical space of each view is limited. For the Task and Event View this means that as the diversity increases, less is available for each line in the visualization. It is already the case that the number of these lines is sometimes too high to be displayed comfortably. Hence the effect of an increase in does not decrease the legibility of these views. Note that the labels for shown in the Task View scale equally poorly as was also identified by Ma and Hellerstein [5]. This can be solved by hiding one or more views.

The height of a bar in the Activity View is distributed over the different task types occurring in the interval represented by that bar. When the number of different task types in one of these intervals increases, this would leave less space for the other task types. The smaller shapes and the amount of them would be less readable and hence make the view more cluttered. On the other hand, it is not likely that the amount of task types increases so much that this becomes an issue.

Concluding, again the overviews are not affected by an increase in the diversity of the data. The detailed views would grow more cluttered as the diversity increases. But it is also possible to (temporarily) hide other views, to yield more vertical space for the remaining views.

### 4.2 Future work

Several ideas floated around that did not make it to the prototype. Either these ideas were out of the scope of the project, or there was not enough time to implement them. This section lists several of those ideas.

**Task hierarchies**

Wafer scanners often perform tasks in parallel. There is a hierarchical relation between these tasks. For instance, when during lot production a problem occurs, a wafer scanner will try to recover from this problem. This so-called auto recovery occurs in parallel to the lot production task but also it is a part of it.

This hierarchy is currently not parsed from the event log. However, the Diagnostics group is developing a technique to derive this hierarchy. This hierarchy could then be visualized in the
current prototype. Ideally, one would like to have a representation of the tasks that reflects this hierarchy. It would be possible to visualize the top-level tasks as they are done now as bars. The sub tasks could be visualized as smaller bars within the top-level task. A schematic example hereof is shown in Figure 4.1a.

The downside of this approach would be that tasks would take more vertical space than they do now. However, this is not a big problem since there would fewer different types of tasks that would be visualized on the top-level, allowing for more vertical space.

Suppose that there would not be enough space to visualize all the tasks in this way. This problem could be solved by following the example of the Productivity View in managing the level of detail using semantic zooming. At the lowest level of detail, one would only see the top-level tasks (Figure 4.1a). One level deeper, the first layer of subtasks could be shown (Figure 4.1b). The next level of detail would also show “subsubtasks” (Figure 4.1c), and so on. The level being displayed would then depend on the vertical space allocated to a task.

On the downside, using semantic zooming in this way would hide some important subtasks such as the auto recovery from the examples in Figure 4.1. In this case there should be a different indication that something else is going on.

Integration
The idea of the overview views was that they could be integrated into existing tools. These views augment each other. So together, they give an overview such that the user can say in a couple of seconds what happened to wafer scanner over a longer period of time.

Together, the overview views take up half of the available screen space. If it were integrated in the current form, it would leave too little space for the rest of the tool. So either these tools would only integrate one of the overviews, or these views would have to be combined.

However, one of the overview views could be integrated in other tools. Therein they could be used as indexes in the data. If needed, it is not difficult to open the other views of the prototype when the user interacts with the embedded view. This can easily be achieved by starting the prototype when requested and feeding it with the needed data.

Field test
Currently, the prototype has not extensively been tested. To properly determine the value of the prototype in the field, more substantial trials have to be run.

Data and process mining
Data and process mining techniques can be used to automatically detect patterns in data and
possibly produce models thereof. There are many usages for these techniques when event logs are considered. Several applications are listed below.

Remember from Section 2.1 that it is difficult to determine which event belongs to which task. So currently, this relation is relaxed such that it only states which events occurred during a task. Since the knowledge of which task produces which events is not readily available, supervised data mining methods are not applicable. On the other hand, for each task we have a set of events that occurred during that task. What needs to be done, is filter all the unrelated events.

So given a number of transactions (in this case tasks), and the goal is to find subsets of these transactions that occur sufficiently often for a specific type of transaction. This is called frequent item set mining. Many algorithms have been developed solve this problem of which the Apriori [13] algorithm is the most commonly used (others include Eclat [14], FP-growth [15] etc.).

This would result in a number of sets of events that occur sufficiently frequent during tasks of a specific type. These sets could then be used to understand which type of events are generated when executing such a task.

There is no one that knows everything there is to know about a wafer scanner. As such they are black boxes to their engineers. To solve such problems, process mining techniques are developed. These techniques model the generation process of events. However, as stated in Section 2.1, not all events are logged. The logging is regulated to reduce the clutter in the log. This regulation makes it difficult to apply process mining techniques, since this would generate a model of the normal functionality of such a scanner, since this would then model the incorrect behavior rather than the expected behavior.
Chapter 5

Conclusions

The exploration of large data sets is an important yet difficult problem, which information visualization techniques may help to solve. Visual data exploration of event logs has many applications, such as fraud detection, that already use information visualization techniques to speed up data analysis.

This thesis describes an interactive prototype that visualizes ASML TWINSCAN event logs. The prototype was designed and built that allows a user to quickly see what a wafer scanner has been doing. When desired, details of specific periods in time are displayed.

The sheer size of the data poses a problem; there is simply no way to visualize all data at the same time. Following Shneiderman's mantra, the data has been abstracted in different levels to ensure that the user is presented with an overview of the data both in terms of productivity as well as activity. When more details are needed, the user can zoom in on the detailed tasks and logged events while maintaining an overview.

To further reduce visual clutter, the Productivity View uses semantic zooming to manage the amount of displayed details. A conceptual application of this semantic zooming technique has been described in relation to the Task View. However, to be able to apply this technique a consistent task hierarchy has to be inferred. While ASML is currently working on some inference, this has unfortunately not yet been completed.

One of the key problems of many applications is the performance. Often applications become too slow when a large amount of data is loaded. To make sure that the developed prototype does not suffer from the same problems, several techniques have been applied. First, the visualized data is cached such that the data does not have be retrieved upon every redraw. Secondly, the data is only redrawn if it absolutely needs to be redrawn. And finally, multiple threads are used to keep the visualization responsive while data is retrieved.

To ensure that the prototype does not run out of memory when the size of a log increases, ten days' worth of logs have been loaded in the prototype. During this test only a couple hundred megabytes of memory was used by the prototype. So in modern computers an increase in the size of the data would not cause a computer to run out of memory.

The developed prototype was designed specifically for ASML's TWINSCAN logs. As a result, the described techniques are not directly applicable to the other event logs described in the academic community. Additionally, the prototype claims the full screen space, making
it difficult to integrate into other tools. However, each of the views can be collapsed giving more space to other components. Furthermore, one or two views can be extracted and added to existing tools. The whole prototype could then be launched when requested.

The prototype has been reviewed by two ASML field engineers. They noted that the tool was useful but it did not suit their needs as they are more concerned with diagnosis. On the other hand, the prototype does provide more context when analyzing a problem and it can be used as a communication tool with management. A more extensive usability test should still be conducted.

The overviews of the prototype could be used as an index into more detailed textual representations of the data. These views could be integrated in the existing tools by ASML. However, since these overviews were designed to work together, some additional work has to be done to combine their information in a single view.
Bibliography


Appendix A

Task types

This appendix lists all types of tasks in Table A.1. The indications of the occurrences have been averaged of three different machines over half a year.

Table A.1: List of all task types. An estimate is given of how often they occur in a day. Additionally the average duration of such a task is given. Gray rows indicate main events and italic rows indicate a type that never occurred in any inspected machine.

<table>
<thead>
<tr>
<th>Type</th>
<th>Above average</th>
<th>Average</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOT</td>
<td>150.54</td>
<td>82.45</td>
<td>0</td>
</tr>
<tr>
<td>IDLE</td>
<td>56.91</td>
<td>212.06</td>
<td>21.66</td>
</tr>
<tr>
<td>PODPLACED</td>
<td>42.7</td>
<td>36.15</td>
<td>0</td>
</tr>
<tr>
<td>PODREMOVED</td>
<td>42.45</td>
<td>35.73</td>
<td>0</td>
</tr>
<tr>
<td>MXACTION</td>
<td>36.97</td>
<td>166.64</td>
<td>0</td>
</tr>
<tr>
<td>WAFERCLEARED</td>
<td>14.28</td>
<td>16.29</td>
<td>1.46</td>
</tr>
<tr>
<td>DATACHANGED</td>
<td>14.05</td>
<td>15.03</td>
<td>24.3</td>
</tr>
<tr>
<td>ADDON</td>
<td>4.8</td>
<td>1.82</td>
<td>4.26</td>
</tr>
<tr>
<td>WTC STONE EXTENDED</td>
<td>4.42</td>
<td>1.41</td>
<td>0</td>
</tr>
<tr>
<td>WTC STONE RETRACTED</td>
<td>4.42</td>
<td>1.41</td>
<td>0</td>
</tr>
<tr>
<td>SX STATUS</td>
<td>4.22</td>
<td>5.25</td>
<td>0.77</td>
</tr>
<tr>
<td>BOARD REBOOT</td>
<td>4.14</td>
<td>2.46</td>
<td>29.02</td>
</tr>
<tr>
<td>WTC</td>
<td>2.73</td>
<td>0.14</td>
<td>0</td>
</tr>
<tr>
<td>TEST</td>
<td>2.53</td>
<td>5.3</td>
<td>9.08</td>
</tr>
<tr>
<td>ALHC EVENT</td>
<td>2.45</td>
<td>102.31</td>
<td>0</td>
</tr>
<tr>
<td>STARTUP</td>
<td>2.1</td>
<td>1.49</td>
<td>9.12</td>
</tr>
<tr>
<td>ADEL REPORT</td>
<td>2.01</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>BOARD DUMP</td>
<td>1.72</td>
<td>0.78</td>
<td>9.66</td>
</tr>
<tr>
<td>DRIVER FAILED</td>
<td>1.41</td>
<td>0.79</td>
<td>8.99</td>
</tr>
<tr>
<td>DAY SUMMARY</td>
<td>1</td>
<td>24:00:00</td>
<td>1</td>
</tr>
<tr>
<td>LOTQUEUESTOPPED</td>
<td>0.98</td>
<td>3.09</td>
<td>0</td>
</tr>
<tr>
<td>LOTQUEUESTARTED</td>
<td>0.88</td>
<td>2.26</td>
<td>0</td>
</tr>
<tr>
<td>RETICLECLEARED</td>
<td>0.82</td>
<td>1.19</td>
<td>0</td>
</tr>
<tr>
<td>TASK DIED</td>
<td>0.81</td>
<td>0.96</td>
<td>4.97</td>
</tr>
<tr>
<td>STARTUP DRIVERS REQUEST</td>
<td>0.79</td>
<td>0.49</td>
<td>0</td>
</tr>
<tr>
<td>IH FLUID ON</td>
<td>0.68</td>
<td>0.27</td>
<td>0</td>
</tr>
<tr>
<td>IH FLUID STABILIZING</td>
<td>0.68</td>
<td>0.27</td>
<td>0</td>
</tr>
<tr>
<td>AUTORECOVERY</td>
<td>0.66</td>
<td>0.37</td>
<td>0</td>
</tr>
</tbody>
</table>
## Appendix A. Task types

Table A.1: List of all task types. (Cont.)

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Above average</th>
<th>Average</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACK STATUS</td>
<td>0.65</td>
<td>0.37</td>
<td>0.19</td>
</tr>
<tr>
<td>AUTORECOVERY</td>
<td>0.65</td>
<td>0.37</td>
<td>0</td>
</tr>
<tr>
<td>FINISHED</td>
<td>0.51</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>IH FLUID STABILIZED</td>
<td>0.34</td>
<td>0.69</td>
<td>0.21</td>
</tr>
<tr>
<td>TESTQUEUESTARTED</td>
<td>0.34</td>
<td>0.69</td>
<td>0.21</td>
</tr>
<tr>
<td>TESTQUEUESTOPPED</td>
<td>0.34</td>
<td>0.69</td>
<td>0.21</td>
</tr>
<tr>
<td>TRACK STATUS</td>
<td>0.32</td>
<td>0.18</td>
<td>3.63</td>
</tr>
<tr>
<td>AUTORECOVERY</td>
<td>0.28</td>
<td>0.23</td>
<td>57.87</td>
</tr>
<tr>
<td>CT NOTOK</td>
<td>0.27</td>
<td>0.35</td>
<td>0</td>
</tr>
<tr>
<td>GASREFILL</td>
<td>0.27</td>
<td>0.35</td>
<td>0</td>
</tr>
<tr>
<td>GASREFILLDONE</td>
<td>0.26</td>
<td>0.17</td>
<td>0</td>
</tr>
<tr>
<td>CONTEXT INFORMATION</td>
<td>0.25</td>
<td>0.17</td>
<td>57.39</td>
</tr>
<tr>
<td>CT OK</td>
<td>0.23</td>
<td>0.18</td>
<td>1.77</td>
</tr>
<tr>
<td>CT REBOOT</td>
<td>0.23</td>
<td>0.18</td>
<td>0</td>
</tr>
<tr>
<td>CT IN</td>
<td>0.2</td>
<td>0.17</td>
<td>1.26</td>
</tr>
<tr>
<td>SYSTEMSTART</td>
<td>0.19</td>
<td>0.17</td>
<td>1.44</td>
</tr>
<tr>
<td>INIT STEP FAILED</td>
<td>0.17</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>SHUTDOWNDRIVERS</td>
<td>0.16</td>
<td>0.09</td>
<td>1.7</td>
</tr>
<tr>
<td>SYSTEMSTOP</td>
<td>0.15</td>
<td>0.09</td>
<td>0.83</td>
</tr>
<tr>
<td>LOTPREPFAILED</td>
<td>0.13</td>
<td>0.09</td>
<td>0.11</td>
</tr>
<tr>
<td>CT NOT INIT</td>
<td>0.23</td>
<td>0.18</td>
<td>0</td>
</tr>
<tr>
<td>CT REBOOT</td>
<td>0.23</td>
<td>0.18</td>
<td>0</td>
</tr>
<tr>
<td>CT OK</td>
<td>0.13</td>
<td>0.05</td>
<td>0</td>
</tr>
<tr>
<td>SYSTEMSTART</td>
<td>0.09</td>
<td>0.04</td>
<td>0</td>
</tr>
<tr>
<td>SHUTDOWNSYSTEM footh</td>
<td>0.07</td>
<td>0.1</td>
<td>0.59</td>
</tr>
<tr>
<td>NOT INITIALIZED</td>
<td>0.05</td>
<td>0.15</td>
<td>0</td>
</tr>
<tr>
<td>FATAL AUTONOMOUS</td>
<td>0.01</td>
<td>0</td>
<td>0.83</td>
</tr>
<tr>
<td>MBDS EVENT</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SW RELEASE CHANGE</td>
<td>0</td>
<td>0</td>
<td>229.11</td>
</tr>
<tr>
<td>SHUTDOWNSYSTEM_UNIX</td>
<td>0</td>
<td>0</td>
<td>0.04</td>
</tr>
<tr>
<td>UNDEFINED</td>
<td>0</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>CUSTOMERMXACTION</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AM UNSTABLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AM STABLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AM METRO FRAME</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AM METRO FRAME STABLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AM ILLUMINATOR</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AM ILLUMINATOR STABLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COVEROPEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COVERCLOSED</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOT QUEUE CONTINUED</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PDMISPLACED</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PREPMAINTENANCE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GASMAINTENANCE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GASMAINTENANCEDONE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TESTQUEUE STARTED EUV</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TESTQUEUE STOPPED EUV</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CT READY</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table A.1: List of all task types. (Cont.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Above average</th>
<th>Average</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT NOTREADY</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>CT ATUNSTABLE</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>CT ATSTABLE</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>CT LTUNSTABLE</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>CT LTSTABLE</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>DATE CHANGE</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>IRIS</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>DAILY MEASUREMENT START</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>WEEKLY MEASUREMENT START</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>MAX TASK TYPE</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>IRIS END</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>LAMP INT CAL END</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>LAMP OFF</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>AUTOM FOCUS CORR START</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>AUTOM SYSTEM CORR END</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>WH PORT COVER OPEN</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>WH PORT COVER CLOSED</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>WS BOARD DUMP</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>RS BOARD DUMP</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>WTC END</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC PREPARE</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC CLEANING</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC RECOVERING</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC FLUSH LB</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC CHUCKSWAP</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC CLEAN IHD</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC CLEAN MB</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC RINSE IHD</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC RINSE MB</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC CLEAN CH1</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC CLEAN CH2</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC FLUSH HOSES</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC RINSE CH1</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
<tr>
<td>ILC RINSE CH2</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
<td>0 00:00:00</td>
</tr>
</tbody>
</table>
Table A.1: List of all task types. (Cont.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Above average</th>
<th>Average</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILC LOW CHEM CS</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC START DISPENSE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC STOP DISPENSE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC LOW CHEMICAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC CHEM CONST HIGH</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC CHEM CONST LOW</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC CHEM Const in spec</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC SS OPENED</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC SS CLOSED</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC BYPASS ACT</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC BYPASS DEACT</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC BYPASS DEACT DELAY</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC DRAIN CHEMICAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC DRAIN NORMAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILC DRAIN NORMAL DELAY</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LASER ENERGY DERATED</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LASER ENERGY NOMINAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WAFER LEVELING NOTIFICATION</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CONFIG NOTIFICATION</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS PLC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS STATE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS TRANSFER SEQUENCE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS FUNCTIONAL SEQUENCE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS MANUAL MODE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS WAITING</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS RGA</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS GRANT</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS STATE MONITORING</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS PUMP</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS ERROR</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS FIRMWARE DOWNLOAD</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS RESET VACUUM PLC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS ERROR UNKNOWN CAUSE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS OPTICS LIFETIME SEQUENCE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OTAS DATA COLLECTED</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LCPLIS DATA COLLECTED</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SPECTRAL PURITY FILTER</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Appendix B

Development

This appendix describes the technical properties of the developed software. Appendix B.1 describes the used techniques, languages and the build method. Appendix B.2 lists changes to the SDT library. Finally, Appendices B.3 and B.4 describe some metrics to indicate the complexity of the software, and the designed architecture respectively.

B.1 Environment

Table B.1 gives a summary of the used techniques. Note that to be as flexible as possible, no additional Java libraries were used.

| Programming language | Java 1.6.0.45-b06, 32 bit |
| Libraries            | ASML’s SDT engine slightly modified to expose internal data structures. See also Appendix B.2. |
| IDE                  | Eclipse |
| Build system         | Ant which internally compiles a windows 32 bit dll library using mingw-32’s Make. |
| OS                   | Windows 7 64 bit |

Table B.1: The programming environment.

The Ant build system has three important targets (which can be run either from the command line using: `ant <target>` or from your favorite IDE):

- build Compiles both the Java as well as the C code
- clean Removes all files created during compilation and distribution.
- fullBuild First cleans all files, then compiles all code and finally constructs a zip file in the dist subdirectory which contains all files needed to run the code.
To be able to use the ant script, a specific directory structure is required as shown in Figure B.1.

Figure B.1: The directory structure needed by the ant build script. If this is not the case, the ant build script has to be adapted.
B.2 Library Changes

ASML provided the SDT engine library. To expose the internal data structure, a few changes were made to the c-java interface. These changes are listed in Listings B.1 and B.2. Red and green coloured lines represent deleted and added lines respectively.

Listing B.1: Changes to the COEVJN.c file.

```c
1671 #ifdef
1672 #endif
1673
1674 JNIEXPORT jint JNICALL Java_com_asml_sdt_engine_Engine_getTaskTypeID( 1675     JNIEnv *env, jobject obj, jint index) {
1676     return COEVDA_GetTask_Type(index);
1677 }
1678
1679 JNIEXPORT jint JNICALL Java_com_asml_sdt_engine_Engine_getWafers( 1680     JNIEnv *env, jobject obj, jint index) {
1681     jintArray wafers = NULL;
1682     const int waferTypes = 4;
1683     COEVDA_TaskTP * task;
1684     COEVDA_LotDetailsTP * dets = NULL;
1685     // Get the corresponding task
1686     result = COEVDA_GetTask(index, &task);
1687     if (result == OK && (task->Type == COEVDA_LOT)) {
1688         // Get the details
1689         dets = (COEVDA_LotDetailsTP *) task->Details;
1690         // Create an array with correct number of elements
1691         wafers = (jintArray)(*env)->NewIntArray(env, waferTypes);
1692         // If unable to allocate, return
1693         if (wafers == NULL) {
1694             return NULL;
1695         }
1696         jint exposed = dets->WafersExposed;
1697         jint remaining = dets->WafersFromTrack + dets->WafersFromCarrier -
1698             dets->WafersToTrack - dets->WafersToCarrier;
1699         jint rejected = dets->WafersRejected;
1700         jint unexposed = dets->WafersUnexposed;
1701         jint waferArray[waferTypes];
1702         memset( waferArray , 0, waferTypes * sizeof(jint) );
1703         waferArray[0] = exposed;
1704         waferArray[1] = remaining;
1705         waferArray[2] = rejected;
1706         waferArray[3] = unexposed;
1707         // Copy the event ids to the output array
1708         (*env)->SetIntArrayRegion(env, wafers, 0, waferTypes, waferArray);
```
Appendix B. Development

> return wafers;
>
> JNIEXPORT jstring JNICALL Java_com_asml_sdt_engine_Engine_getEventDateTime (
> JNIEnv *env, jobject obj, jint index) {
> COEVDA_LogEntryTP *event;
> int result = COEVDA_Get_LogEntry(index, &event);
> if (result == OK) {
>     char dateTime[COEVDA_DATE_SIZE + COEVDA_TIME_SIZE] = "";
>     if (dateTime != NULL) {
>         sprintf(dateTime,"%s %s", event->Date, event->Time);
>         jstring jDateTime = (jstring)(jint)NewStringUTF(env, dateTime);
>         return jDateTime;
>     }
> }
> return (*env)->NewStringUTF(env, "***Error***");
>
> JNIEXPORT jstring JNICALL Java_com_asml_sdt_engine_Engine_getEventSeverity(
> JNIEnv *env, jobject obj, jint index) {
>     COEVDA_LogEntryTP *event;
>     int result = COEVDA_Get_LogEntry(index, &event);
>     if (result == OK) {
>         char *severity = event->Severity;
>         if (severity != NULL) {
>             jstring sev = (*env)->NewStringUTF(env, severity);
>             return sev;
>         }
>     }
>     return (*env)->NewStringUTF(env, "***Error***");
> }
>
> JNIEXPORT jint JNICALL Java_com_asml_sdt_engine_Engine_getTaskIndent(
> JNIEnv *env, jobject obj, jint index) {
>     int result = OK;
>     COEVDA_TaskTP *task;
>     result = COEVDA_Get_Task(index, &task);
>     if (result != OK) {
>         return (jint)-1;
return ((( jint ) task )--IndentCount );
}

/*--------------------------------------------------------------*/
JNICALL Java_com_asml_sdt_engine_Engine_getTaskReticles
(JNIEnv *env, jobject obj, jint taskID ) {
int result = OK;
int i;
COEVDATaskTP * task;
jobjectArray ret = NULL;
COEVGN_StrListTP * reticles ;
COEVDALotDetailsTP * details ;

result = COEVDAGetTask(taskID, &task );

if ( result == OK && task->Type == COEVDALot ) {
details = (COEVDALotDetailsTP *) ( task->Details );
reticles = details->ReticlesUsed ;

ret = (jobjectArray) (*env)->NewObjectArray(
    env ,
    reticles->Count ,
    (*env)->FindClass(env, "java/lang/String" ),
    NULL );

for ( i = 0; i < reticles->Count ; i++ ) {
    (*env)->SetObjectArrayElement(
        env ,
        ret ,
        i ,
        (*env)->NewStringUTF(
            env ,
            reticles->Items[i] )
    );
}

return ret ;
}
Listing B.2: Changes to the Engine.java file.

```
1291 c1291, 1339
< }
\ No newline at end of file

1291 */
> public native int[] getWafers(int taskID);

1291 */
> public native String getEventDateTime(int eventID);

1291 */
> public native String getEventSeverity(int eventID);

1291 */
> public native int getTaskIndent(int taskID);

1291 */
> public native String[] getTaskReticles(int taskID);
```
Appendix B. Development

Table B.2: The code metrics per package. Note that some classes directly reside in the Data and UI packages. Hence the aggregations of the metrics of these packages do not directly follow from the metrics of their sub-packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Lines of code</th>
<th>Comment lines of code</th>
<th>JavaDoc method coverage</th>
<th>Average cyclomatic complexity of methods</th>
<th>Number of classes</th>
<th>Number of methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drawable</td>
<td>539</td>
<td>192</td>
<td>42.4%</td>
<td>1.42</td>
<td>5</td>
<td>59</td>
</tr>
<tr>
<td>Enums</td>
<td>543</td>
<td>120</td>
<td>69.8%</td>
<td>1.91</td>
<td>7</td>
<td>23</td>
</tr>
<tr>
<td>Filter</td>
<td>199</td>
<td>56</td>
<td>30.0%</td>
<td>1.75</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>Structures</td>
<td>552</td>
<td>152</td>
<td>85.7%</td>
<td>3.25</td>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>2573</td>
<td>931</td>
<td>64.2%</td>
<td>1.85</td>
<td>23</td>
<td>176</td>
</tr>
<tr>
<td>Engine</td>
<td>1746</td>
<td>1150</td>
<td>98.5%</td>
<td>2.56</td>
<td>7</td>
<td>135</td>
</tr>
<tr>
<td>Main</td>
<td>197</td>
<td>3</td>
<td>0%</td>
<td>1.50</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>UI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Custom</td>
<td>1487</td>
<td>114</td>
<td>15.1%</td>
<td>1.60</td>
<td>41</td>
<td>112</td>
</tr>
<tr>
<td>Language</td>
<td>64</td>
<td>26</td>
<td>100%</td>
<td>1.33</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>View</td>
<td>3395</td>
<td>616</td>
<td>32.1%</td>
<td>2.52</td>
<td>53</td>
<td>215</td>
</tr>
<tr>
<td></td>
<td>5013</td>
<td>791</td>
<td>28.7%</td>
<td>2.17</td>
<td>96</td>
<td>336</td>
</tr>
<tr>
<td>Grand Total</td>
<td>9528</td>
<td>2875</td>
<td>55.95%</td>
<td>2.18</td>
<td>132</td>
<td>654</td>
</tr>
</tbody>
</table>

B.3 Metrics

Some metrics of the developed software can be found in Table B.2 aggregated per package. These metrics provide an indication of the complexity and maintainability of the software. For instance, files with thousands upon thousands of lines of code are considered to be unmaintainable. The cyclomatic complexity basically represents the number of different execution paths in the code. The general idea is that methods or functions with a cyclomatic complexity greater than ten are considered to be unmaintainable. All other metrics are rather straightforward.

B.4 Architecture

This Appendix explains a little more about the architecture of the developed software. It has been divided into three main packages: data, engine and ui (see also Figure B.2).

The data package contains classes pertaining to the data used in the application. The drawable sub-package contains classes that describe the data. It contains an abstract base class DrawableData, which describes some generic methods on data, which can be drawn. Three subclasses thereof can be used to describe events, tasks and an abstraction thereof. Furthermore the data package contains a number of enums which describe several constants of the data, for instance which types of tasks can be found in the logs.

The engine package deals with the interaction with the engine. It contains the (slightly modified) Engine class describing the JNI interface with the native c library that parses the log files. The EngineWorker class is a wrapper for the Engine class. This class is mainly used to abstract from Engine specific details and is able to generate the application specific data
structures as described in the data package.

Finally, the ui package is the more convoluted one. It hosts the custom sub-package that is basically a place in which all custom ui classes (not directly pertaining to the visualizations) are stored. An example of this would be the FileDialog class which is used to select the log files which have to be read by the engine. Furthermore, it also contains the language sub-package. This package contains a class that abstracts from the Java’s ResourceBundle. It is used to ease localization for all user interface related strings.

The most important sub-package is the view package. This contains all classes representing the actual visualizations. However, as the visualizations share a large part of their functionality, this was encapsulated in the abstract base class TimePanel. This class controls all interactions, as well as the functionality to draw the axis and other useful functions (such as a function which determines the time on a specific pixel). These methods in turn are used by the subclasses BarView, RiverView, TaskView and EventView which are intricately dependent on one another. The RiverView name was originally used for the ThemeRiver-like [11] implementation of the Activity view (Section 3.3).
Figure B.2: The package structure of the application.