Parallelizing test, diagnose and fix tasks using graph partitioning algorithms

de Jong, I.S.M.; Boumen, R.; van de Mortel - Fronczak, J.M.; Rooda, J.E.

Published: 01/01/2007

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Dec. 2018
SE Report: Nr. 2007-11

Parallelizing test, diagnose and fix tasks using graph partitioning algorithms

I.S.M. de Jong, R. Boumen,
J.M. van de Mortel-Fronczak and J.E. Rooda

ISSN: 1872-1567

SE Report: Nr. 2007-11
Eindhoven, February 2008
SE Reports are available via http://se.wtb.tue.nl/sereports
Abstract

The development of a new semi-conductor manufacturing system, like the ASML waferscanner, is mainly driven by time-to-market. The final task in the development phase of a wafer-scanner is a test-diagnose-fix task containing many (100+) test cases. The duration of this test-diagnose-fix task can be reduced by using an additional test system of the same configuration and execute the test cases in parallel. The set of test cases to be executed needs to be divided into two sets of test cases. The duration of the test tasks that are executed in parallel should minimal.

Several approaches exist to divide test cases into two sets of test cases. The standard bin-packing approach does not take into account that certain test cases ‘overlap’ each other, while this overlap could influence the total test duration in a positive and negative manner.

In this paper, an algorithm is presented which partitions a set of test cases into two smaller sets of test cases, such that the minimal total test duration is obtained when these test sets are executed in parallel. The algorithm is based on a well-known graph partitioning algorithm, which is adopted such that test cases (test graphs) can be partitioned. The algorithm uses an intuitive system test model as input and takes the effect of overlapping test cases, test stop criteria, the test sequence and the test process into account. The case on the ASML system shows that the duration of a test task is reduced by 30% by parallelizing this task.
Parallel execution of test cases on two test systems can be the only option to meet the deadline in a time-to-market driven development environment, like the development of a new type of ASML wafer scanner. Purchasing an additional test system is considered for each test-diagnose-fix task on the critical path. Parallel execution reduces the duration of a test-diagnose-fix task with approximately 50%, depending on the duration of the individual test cases.

Dividing test cases over two test-diagnose-fix tasks can be done manually for small and simple test sets or with existing bin-packing algorithms [1] for larger test sets with different test durations per test case. The overlap of test cases is not taken into account in these algorithms, only the test duration per test case is taken into account. It is shown in [2] that this coverage information can be beneficial in terms of total test duration by selecting the optimal test sequence. The reduction of test time can be up to 20 – 30% if the coverage info is taken into account to determine a better test sequence. Executing a test-diagnose-fix task in parallel itself leads to a reduction of the duration of roughly 50%. An additional duration reduction of 20 – 30% per executed test-diagnose-fix task in parallel is expected, when the coverage information from the test cases is used. The total expected reduction of parallelizing test-diagnose-fix tasks in combination with the usage of the coverage information is therefore in the range of 60 – 65%.

A set of test cases can be divided into two smaller sets of test cases, such that these smaller sets can be executed in parallel or in series. A set of test cases can be divided into two smaller sets of test cases, such that these smaller sets can be executed in parallel or in series. The advantage of executing two sets in parallel is that the duration of the test-diagnose-fix task can be reduced by more than a factor two. Dividing the test cases into two sets of test cases that are executed in series can be used as a heuristic for a sequencing algorithm as introduced in [2]. Only parallel execution is considered in this paper. A simple and straightforward division of test cases for parallel execution takes only the test duration into account. Test cases are divided such that two sets of test cases are obtained and the maximum duration of the two test sets of test cases is minimal. The duration of the set of test cases is determined by summing up the duration of the individual test cases in each test set. Several algorithms are available to solve this problem. However, the resulting test sets are equal in test duration if and only if all test cases result in a pass verdict, no faults are present in the system and no diagnosis and fix tasks are required. In practice, test cases fail and diagnosis and fixes are required. The diagnosis and fix duration, and the test sequence, influence the test duration. In fact, the total test duration $\Phi$ of a test set depends on the test cases, the test sequence, the actual set of problems in the system under test and the test stop criteria [3]. Hence, these four aspects need to be taken into account while partitioning a set of test cases.

Graph partitioning algorithms are used in our approach to partition a set of test cases into smaller sets of test cases. To be able to use this approach, we need to translate a set of test cases into a test graph. This is done in two steps. First, the set of test cases is represented as a system test model where test cases are related to possible faults in the system and relevant properties for test cases and fault states are given. Second, the system test model is translated into a test graph. The test graph is partitioned using a multi-level hypergraph partitioning method. This method originates from the domain of parallel computing. The test graph is partitioned using a multi-level hypergraph partitioning method. This method originates from the domain of parallel computing [4]. The objective of partitioning in the domain of parallel computing is reducing the duration of a large calculation. The durations of the executed tasks can be summed up. In the test domain, the cost (test duration or cost) depends on the test cases, the test sequence, the actual system under test and the test stop criterion. Consequently, the total test duration is not a summation of the test case durations, but requires a more advanced calculation. Therefore, cost estimators have been developed.

This paper is organized as follows: first, the system test model is introduced in Section 2. Section 3 is a general introduction into graph partitioning. The graph partitioning section is...
followed by Section 4 on system test model partitioning, where the details on partitioning the system test model are introduced as well as the adaption to the available graph partitioning algorithm. The objective functions and estimators are explained in Section 5. An example and an industrial case study are presented in Sections 6 and 7. In Section 8 conclusions and future work are presented.

2 System test model

The system test model used for partitioning test-diagnose-fix tasks is explained in this section. The system test model is based on the basic test model used for diagnosis sequencing [5]. Boumen [2] uses a system test model, based on the basic test model, for probabilistic test sequencing. The test model of Boumen is extended with additional properties, such that the resulting system test model can be used for partitioning test-diagnose-fix tasks. The basic test model [5] is defined as \( D = (T, S, C, P, R) \). Our system test model \( D \) is defined as a nine-tuple \((T, S, (C_T, \varphi_T), (C_D, \varphi_D), (C_F, \varphi_F), (C_AF, \varphi_AF), P, I, R_0)\), where:

- \( T \) is a finite set of \( k \) tests.
- \( S \) is a finite set of \( l \) fault states.
- \( C_T : T \to \mathbb{R} \) gives for each test in \( T \) the associated cost of performing that test.
- \( \varphi_T : T \to \mathbb{R} \) gives for each test in \( T \) the associated duration of performing that test.
- \( C_D : T \to \mathbb{R} \) gives for each test in \( T \) the associated cost of diagnosing the failed test.
- \( \varphi_D : T \to \mathbb{R} \) gives for each test in \( T \) the associated duration of diagnosing the result of the test if the result is \textit{fail}.
- \( C_F : S \to \mathbb{R} \) gives for each fault state that is to be fixed the cost of fixing that fault state.
- \( \varphi_F : S \to \mathbb{R} \) gives for each fault state in \( S \) the associated duration of developing a fix for the fault state.
- \( C_AF : S \to \mathbb{R} \) gives for each fault in \( S \) the associated cost of applying the fix on the system under test.
- \( \varphi_AF : S \to \mathbb{R} \) gives for each fix that is to be applied the associated duration.
- \( P : S \to \mathbb{R} \) gives for each fault state in \( S \) the \textit{a priori} probability that the fault state is present, which is the absolute probability that a certain fault is present.
- \( I : S \to \mathbb{R} \) gives for each fault state in \( S \) the impact of the fault state if the fault state exists in the system under test. Impact can be a value relative to each fault state or a specific value related to a property of the system under test.
- \( R_0 : T \times S \to \mathbb{R} \) gives for each test \( t \) and fault state \( s \) the coverage of the test \( t \) on fault state \( s \) in the range of 0 to 1.

As an example, a system test model is defined for a common telephone. The telephone consists of a handset, a cable and a device. Each of these modules can contain a fault. The interfaces between the modules can also contain one fault. Hence, in total, 5 fault states are defined. Additionally, 6 test cases are defined covering the 5 fault states. A graphical view of the telephone is given in Figure 1. The set \( S \) of five possible fault states in the telephone is:

1. \( s_1 \) the device contains a fault
2. \( s_2 \) the cable contains a fault

3 System test model
The set $T$ of six tests that are available to cover these fault states is:

1. $t_0$ the test of the complete phone system
2. $t_1$ the test of the device
3. $t_2$ the test of the cable
4. $t_3$ the test of the handset
5. $t_4$ the test of the device and the cable
6. $t_5$ the test of the handset and the cable

A matrix representation of the system test model $D$, including the properties per fault state and test, is given in Table 1.

<table>
<thead>
<tr>
<th>$S / T$</th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$P$</th>
<th>$\varphi_T$</th>
<th>$C_T$</th>
<th>$C_{AF}$</th>
<th>$\varphi_{AF}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>0.2</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0.3</td>
<td>0</td>
<td>10%</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$s_2$</td>
<td>0.2</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.3</td>
<td>0.3</td>
<td>10%</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$s_3$</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.3</td>
<td>10%</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$s_4$</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.3</td>
<td>0</td>
<td>10%</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$s_5$</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.3</td>
<td>10%</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

$C_T$ = 3 1 1 1 2 2
$\varphi_T$ = 6 2 2 2 4 4
$C_D$ = 10 1 1 1 6 6
$\varphi_D$ = 10 1 1 1 6 6

Table 1: A test model for the telephone example

The relation $R_{ts}$ between tests and fault states is represented by a value between 0.0 and 1.0. A coverage of 0.0 indicates that fault state $s$ is not covered by test $t$ and a coverage of 1.0 indicates that it is fully covered, i.e., the test case always detects the fault if it exists in the system. Existing sets of test cases are used as the initial set of test cases. Possible fault states are determined using existing test case knowledge, requirements of the system and FMEA/FMECA analysis. Fault probabilities are estimated and obtained from existing failure databases and historical project data.

The system test model is used for test partitioning in this paper. The next section serves as an introduction into matrix partitioning, followed by Section 4 on system test model partitioning using an hypergraph partitioning algorithm.
3 Matrix partitioning

Originally, partitioning algorithms were used to partition computing jobs over multiple processors. For this purpose, jobs and inter-job communication are represented using a sparse matrix. The sparse matrix is partitioned over the processors, such that the computing time and the communication of data between processors is minimized. Usually, matrix partitioning is formulated as a graph partitioning problem that is NP-hard [6]. The standard approach to partitioning graphs is explained in [4], including the discussion of the shortcomings of this approach. Available software packages like PaToH [7] and hMetis [8,9] implement the partitioning algorithm using heuristics to generate solutions that approach the optimal solution in reasonable time. The standard partitioning problem has been extended with bipartite graph models and hypergraph models. Bipartite graph models allow the modeling of problems where the initial vertices are different from the final vertices. Hypergraph models are a more general form of bipartite models [4].

A hypergraph \( G = (V, H) \) consists of a set \( V \) of vertices and a set \( H \) of hyperedges. Each hyperedge consists of a subset of vertices. In this way, the dependencies are represented as hyperedges and the partitioning goal is to equally distribute the vertices and minimizing the hyperedges that cross the resulting partitions. The hypergraph model is used in this paper. Multi-level partitioning algorithms are proposed in the early 1990s by several researchers, such that large graphs could be partitioned faster. This approach has proven to be quite robust. The general idea behind the approach is as follows:

1. A large graph is grouped into a smaller graph, while preserving the essential properties of the original graph. This process is repeated up to a certain level.

2. The smallest graph is partitioned using a partitioning algorithm.

3. The partition is propagated back through the sequence of larger graphs and being refished along the way.

PaToH, hMetis and Mondriaan are three graph partitioning algorithms that follow this hypergraph approach. The performance of these algorithms are compared using the Rutherford-Boeing sparse matrix Collection [10] in [11]. The purpose of each algorithm is different, while the general performance of the algorithms is comparable. In our case, the generic hypergraph partitioning algorithm is applied for partitioning test-diagnose-fix tasks. For this purpose, the test problem needs to be transformed into a hypergraph model. Second, a specific objective function needs to be defined and implemented using one of the available implementations. The following sections describe the chosen model and algorithms for the partitioning of test-diagnose-fix tasks.

4 System test model partitioning

The approach taken for system test model partitioning is similar to the general partitioning approach. The relations between test cases and fault states are translated into a test graph. This test graph is coarsened into a hypergraph. This hypergraph is partitioned using a local search partitioning algorithm with a dynamic objective function. The reason for the dynamic objective function is that the cost of the test-diagnose-fix task in the partition depends on the coverage of the test cases in that partition. Finally, the matrix is uncoarsened at each coarsening level. The details of each step in this process are explained below.

4.1 The system test model represented as test graph

The system test model \( D \) is translated into a test graph \( G = (V, E) \). The vertices in the test graph are the test cases from the system test model:

\[
v = T
\]
The edges in graph $G$ represent the relations between test cases and fault states. An edge is a pair of a test case and a fault state where the coverage of the test case on a fault state is $> 0$. These pairs follow from the relation $R_G$:

$$E = \{(t, s) \in T \times S \mid R_G(t, s) > 0\}$$

(2)

Using these definitions, a mathematical formulation of a graph partitioning problem is as follows:

**Given:** A directed graph $G = (V, E)$, $K \in \mathbb{N}$, $L \in \mathbb{N}

**Find:** A partition $P = (V_1, ..., V_L)$ of $V$ with $|V_i| \leq K$

for each $1 \leq i \leq L$, with the minimal cost $J_P$.

The minimal cost $J_P$ for $L$ partitions that are run in parallel is the minimum of the maximum cost for each partition $P$ in the set of all possible partitions $\mathcal{P}$:

$$J_P = \min_{P \in \mathcal{P}} \max_{1 \leq l \leq L}(J_l(V_i))$$

(3)

Graph partitioning algorithms are able to find partitions for traditional applications, like processor load balancing, while minimizing communication volume. The traditional graph partitioning algorithm is explained first, starting with graph coarsening.

### 4.2 Graph Coarsening

The graph is coarsened into a hypergraph $H$ consisting of a set of vertices $V$ and a set of hyperedges (nets) $N$: $H = (V, N)$. A set of vertices $V_i$ consists of vertices $V_t$. Each element $v$ of vertex $V_t$ is only present once. So, $V_i \cap V_j = \emptyset$ holds for each vertex in $V$. All elements in the original set of vertices $V$ can be found in the set of vertices $V$: $\bigcup_i V_i = V$. The initial hypergraph $H_0$ consists of all vertices and edges of the original test graph. The initial hypergraph is coarsened into a sequence of smaller hypergraphs $H = \{V_1, N_1\}, H_2 = \{V_2, N_2\}, ..., H_m = \{V_m, N_m\}$. Each coarsened hypergraph is smaller than the previous in the sequence: $|V_2| > |V_3| > ... > |V_m|$, where $V_1$ is the set of uncoarsened vertices: level 0, $V_t$ is the first coarsening level and so on. Coarsening is achieved by combining disjoint subsets of vertices from the hypergraph. This way, multiple vertices form a single vertex in the hypergraph on the next level. The (combined) vertices in the next level can be combined again into a new vertex for the next level. This process stops when the number of the vertices in the hypergraph is below a user defined limit or the ratio between the previous number of vertices and the new number of vertices is below a user defined stop ratio. Graph coarsening is performed using a matching-based clustering technique described in [12] as *Heavy Connectivity Matching*. This technique combines highly connected vertices into combined vertices. Highly connected vertices for system test models are vertices where test cases and fault states are highly connected. The *Heavy Connectivity Matching* algorithm visits vertices in a random order. The highest net connectivity is calculated for all unmatched vertices $v$ in $V$ and the vertices that are adjacent to $v$. Two vertices are adjacent if both vertices share the same net. A net is defined as a set of fault states in a test graph. Two vertices $v$ and $w$ share a net if $\text{nets}(v) \cap \text{nets}(w) \neq \emptyset$, where $\text{nets}(v)$ returns the set of nets connected to vertex $v$. Vertices that share the same net are merged into a vertex $V_t$. Vertex $v$ remains single if no adjacent unmatched vertices exist for vertex $v$. The highest net connectivity is the maximal net connectivity: $|\text{nets}(v) \cap \text{nets}(w)|$. Other matching criteria and strategies can be used. See [12] for a strategy called *Heavy Connectivity Clustering*, that is similar to *Heavy Connectivity Matching*. A vertex in *Heavy Connectivity Matching* is joined with only one other vertex (or none). A vertex in the *Heavy Connectivity Clustering* strategy can be joined with another vertex or with another group of vertices. In this case, no singleton vertices remain at the end. Which vertex or group of vertices is selected for joining a vertex is determined by a selection criterion. The selection criterion for the *Heavy Connectivity Clustering* strategy is based on the connectivity of the newly formed cluster divided by the weight of the newly formed cluster. The weight resembles the number of vertices in the newly formed cluster. The division by the weight...
is required to avoid the polarization towards very large clusters. See [12] for details on the calculation of the weight and connectivity for the newly formed cluster.

4.3 Hypergraph partitioning

Hypergraph partitioning can be performed in one or two dimensions. hMeTiS and PaToH are two software packages that implement one dimensional partitioning algorithms [7, 12]. hMeTiS uses an initial (random or guided) partition that is refined by the Kerninghan-Lin (KL) algorithm [13] followed by the Fiduccia-Matheyse (FM) algorithm [14]: the HKLFM-algorithm (hypergraph Kerninghan-Lin Fiduccia-Matheyse algorithm. PaToH uses a Greedy Hypergraph Growing algorithm [7] for partitioning. A two-dimensional algorithm has been proposed by Vastenhouw and Bisseling [15] and implemented in the Mondriaan software package. The two-dimensional algorithm partitions the vertices as well as the edges in an alternating algorithm where first the vertices are partitioned followed by a partitioning of the edges. The two-dimensional algorithm partitions the vertices as well as the edges in an alternating algorithm where first the vertices are partitioned followed by a partitioning of the edges (or the other way around). The two-dimensional algorithm uses the HKLFM algorithm for each partitioning step in each of the two directions. Mondriaan can also operate as a one-dimensional partitioning algorithm. We use the HKLFM algorithm as partitioning algorithm and the Mondriaan implementation as starting point for test graph partitioning. The objective function, which is specific for test-diagnose-fix tasks, is developed using the Mondriaan implementation.

The hypergraph Kerninghan-Lin Fiduccia-Matheyse (HKLFM) algorithm is a refinement algorithm. First an initial partition is randomly selected. This random initial partition is then refined using the HKLFM algorithm. A single pass of the KLFM algorithm visits all vertices in the graph. Two vertices from different partitions are selected and the cost is calculated assuming that these two vertices were swapped between partitions. The two vertices that result in the maximal gain are selected and these two vertices are placed in each others partitions (the two vertices are swapped). This process repeats for all vertices that are not swapped and stops when no vertices are available for swapping. The entire process can be repeated with the resulting partitions as input. Typically, a small number of passes is required to converge. The algorithm can be restarted a number of times with a completely different randomly selected initial partition. A modification to the KL algorithm, proposed by Fiduccia-Matheyse, improves the run time significantly without decreasing its effectiveness. In this algorithm, only one vertex is moved between partitions with each pass instead of swapping pairs. The gain of moving a vertex is calculated for each vertex in both partitions. The vertex with the highest gain is moved to the other partition and then excluded for further movement in this pass. A vertex is only moved when it does not violate the imbalance constraint as set by the user. The pass ends when no vertices can be moved, that is when no vertices are available to move or no vertex is able to move because of violation of the imbalance constraint. Again, this process can be repeated a number of times with the previous partition as input, such that a better result is obtained.

4.4 Uncoarsening and refinement

The coarsened partitioned hypergraph is uncoarsened in this phase. Uncoarsening means that the coarsened vertex is projected back to a set of vertices. A refinement step is performed after each uncoarsening step. The refinement step is again performed by the HKLFM algorithm using the uncoarsened partitions. This uncoarsening process is repeated until no coarsened vertices are present in the graph, i.e., level 0 is reached.

4.5 Parameters in the HKLFM algorithm

A number of parameters can be set to control the HKLFM algorithm. These parameters are discussed in this subsection. The first parameter, the minimal number of vertices (test cases) in a hypergraph $V_{\text{min}}$ controls the allowed depth of the coarsening phase. The number of vertices in a hypergraph should exceed the minimal number of vertices: $|V| > V_{\text{min}}$. Coarsening is stopped when the minimal number of vertices is reached. More coarsening levels lead to
more executions of the KLFM algorithm. The optimal setting depends on how connected the graph is. The second parameter, the fraction of reduction of the number of vertices $\eta_{\text{red}}$, is used to stop coarsening if subsequent coarsening phases are not beneficial enough. Coarsening is not beneficial enough if the number of vertices between two coarsening phases, $l - 1$ and $l$, is not reduced enough. This parameter is expressed as a fraction: $\frac{|V_{l-1}|}{|V_l|} > \eta_{\text{red}}$. The default setting of $\eta_{\text{red}} = 0.2$ was sufficient in our experiments.

The third and fourth parameters stop the heavy connectivity matching clustering method. The third parameter, $\text{MaxNrVtxInCluster}$, stops clustering when the maximal number of vertices (test cases) in a cluster $C$ is reached:

$$|C| \leq \text{MaxNrVtxInCluster}$$

(4)

This way, the size of the cluster can be controlled and also the depth of the coarsening phase (depending on how the graph is connected). The fourth parameter, $C\text{MaxWghtFrac}$, stops clustering when the weight of a cluster is reached:

$$\frac{\sum_{v \in C_j} \varphi(v)}{\sum_{v \in V_l} \varphi(v)} \leq C\text{MaxWghtFrac}$$

(5)

Where, $C_j$ is the $j$-th cluster in vertex $V_l$ on the $i$-th coarsening level. The weight of a cluster is approximated by the sum of the test durations in the cluster. This way, the size of the cluster in terms test duration (weight) can be controlled such that equally sized clusters are obtained.

### 4.6 Partition cost and communication volume

The original partitioning algorithms are developed to distribute calculations over multiple processors such that the communication volume between processors is minimized and the load on each processor is balanced. A fixed weight for communication overhead is used and the amount of calculations determines the processor load. Communication volume for partitioning system test graphs is similar. The overlap (communication) between test-diagnose-fix tasks should be minimized, because this is beneficial for the duration, cost and remaining risk of the combined test-diagnose-fix task. The main difference between partitioning for parallel distribution of processor load and parallel execution of test cases is how the cost of a partition is determined. The main difference between partitioning for parallel distribution of processor load and parallel execution of test cases is how the cost of a partition is determined. For the distribution of computations, the sum of the tasks is equal to the load on the processor. In other words, the tasks are assumed to be independent of each other. This is not the case for test cases. For example, if two test cases cover the same fault state and the first test case fails because this fault state is actually in the system, then the second test case will also fail. The two test cases therefore depend on each other. The risk reduction and remaining risk in a test-diagnose-fix task depends on the executed test sequence. Test cases with very limited coverage on the fault states in a partition require much more time to reach the remaining risk target or cannot meet the target at all. Hence, the cost of a partition is not a simple summation of the test duration.

The graph partitioning algorithm from literature (HKLFM using the Mondriaan implementation) has been adapted such that the dynamic objective function is used to determine the ‘partition cost’. The Mondriaan toolset is used as implementation instead of PaToH or hMetis, because partitioning fault states could also be beneficial. This extension can be developed easily using Mondriaan, because the basis of Mondriaan is a bipartition algorithm. The developed objective functions are explained in detail in the next section, followed by a small example in Section 6.
5 Objective functions

The objective function of the combined test-diagnose-fix task is expressed in terms of the test duration $\Phi_T$, the cost $C_T$ and the remaining risk after testing $R_{R,T}$. This is also the case for the objective function of a partitioned test-diagnose-fix task as depicted in Figure 2. Figure 2 shows the test duration, cost and remaining risk of the two partitioned test-diagnose-fix tasks $T_1$ and $T_2$ and the test duration, cost and remaining risk of the combined test-diagnose-fix task $T$. The system is copied first in the cpy task, such that test cases can be executed in parallel in the tdf tasks. The results are assembled in the asm task and the integration and test sequence can be continued. This assembly task assembles the risk in the system and is required to determine the complete duration of the parallel test-diagnose-fix task. The objective function of a test-diagnose-fix task is described as vector $\xi$ in Equation (6). Each element in the cost vector $\xi$ consists of two parts that are summed up. The first part is the expected value or maximum value of the test duration, cost or remaining risk. This way it is possible to optimize on the expected value or maximum value or combinations of these. The second part is a penalty function $P$ used to penalize a solution which constraints are not met. Note that the solution is not excluded from the solution set to allow the algorithm to overcome local optima. The penalty function is defined in Equation (8) for the expected test duration. The penalty functions for the other parameters are defined in a similar fashion. The objective function of the test-diagnose-fix task is described in terms of a weight vector $w$. The elements of vector $w$ in Equation (7) describe how important each of the performance measures are. The weight vector is later used in the objective function.

$$\xi = \begin{pmatrix} E(\Phi) + P_E(\Phi), & \Phi_{\text{max}} + P_{\Phi_{\text{max}}}, \\ E(C) + P_E(C), & C_{\text{max}} + P_{C_{\text{max}}}, \\ E(R_R) + P_E(R_R), & R_{R,\text{max}} + P_{R_{R,\text{max}}} \end{pmatrix}$$  \hspace{1cm} (6)

$$w = \begin{pmatrix} w_E(\Phi), & w_{\Phi_{\text{max}}}, \\ w_E(C), & w_{C_{\text{max}}}, \\ w_E(R_R), & w_{R_{R,\text{max}}} \end{pmatrix}$$  \hspace{1cm} (7)

$$P_{(E\Phi)} = \begin{cases} 10^5(E(\Phi, \text{target}) - E(\Phi))^2 & \text{if } E(\Phi) > E(\Phi, \text{target}) \\ 0 & \text{if } E(\Phi) \leq E(\Phi, \text{target}) \end{cases}$$  \hspace{1cm} (8)

The objective function of a single test-diagnose-fix task is defined according to Equation (9). The weight function typically contains 0’s and a single 1 to indicate which performance indicator should be optimized. More complex weight vectors are possible, to accommodate a trade-off between performance indicators.

$$J = \xi \cdot w^T$$  \hspace{1cm} (9)
The performance of the combined test-diagnose-fix task is also expressed in terms of test duration $\Phi_T$, cost $C_T$ and remaining risk $R_{R,T}$. The expected cost $E(C_T)$ is calculated using Equation (10).

$$E(C_T) = E(C_{T_1}) + E(C_{T_2})$$

The test duration of the combined test-diagnose-fix task is the maximum of the test durations of the parallel test-diagnose-fix tasks:

$$\Phi_T = \max(\Phi_{T_1}, \Phi_{T_2})$$

Note that this is a maximum of two discrete variables $\Phi_{T_1}$ and $\Phi_{T_2}$. The maximal test duration of the combined test-diagnose-fix task is determined using Equation (12). Here, the maximum duration is the maximum of the maximal test duration of both parallel test-diagnose-fix tasks.

$$\Phi_{T,max} = \max(\Phi_{T_{1, max}}, \Phi_{T_{2, max}})$$

The risk in the system is the summed product of the a-priori failure probability and impact for each fault state in the system:

$$R_{R,T} = \sum_{s \in S} P(s) I(s)$$

Executing test case $t$ reduces the failure probability of a fault state $s$ by $1 - R_0(t, s)$. Therefore, executing test cases reduces the risk of certain fault states. The reduction of the failure probability due to test sequence $G$ is expressed by:

$$\Delta_P(s, G) = 1 - \prod_{t \in G} (1 - R_0(t, s))$$

The calculation of the remaining risk in the combined test-diagnose-fix task is more involved, because a certain fault state can now be present in two (or more) parallel test-diagnose-fix tasks. Executing test cases in both parallel test-diagnose-fix tasks reduces the risk of the fault state in both test-diagnose-fix tasks. The reduction of the risk in both test-diagnose-fix tasks should be taken into account, because the combined remaining risk is lower, hence, the remaining risk stop criterion could be reached earlier. Therefore, the remaining risk of the combined test-diagnose-fix task depends on the fault states and the sequences executed in each of the parallel test-diagnose-fix tasks. The fault states in the parallel test-diagnose-fix tasks can be overlapping, i.e., one fault state can be covered by two test-diagnose-fix tasks.

The set of fault states $S$ is split up into subsets to determine the remaining risk and maximal remaining risk for the parallel test-diagnose-fix tasks. Three subsets can be distinguished for a test-diagnose-fix task with two parallel test-diagnose-fix tasks: $S_1$, $S_2$ and $S_o$. Where, the subset $S_1$ are the fault states that are covered in test-diagnose-fix task 1 and $S_2$ are the fault states covered in test-diagnose-fix task 2. The overlapping fault states, $S_o$, are the fault states that are covered in both test-diagnose-fix task 1 and 2: $S_o = S_1 \cap S_2$.

The executed test sequence is of importance, because the sequence determines how much risk is reduced during testing [3]. The remaining risk of the combined test-diagnose-fix task therefore depends on the test sequences of both parallel test-diagnose-fix tasks, $G_1$ and $G_2$, and the fault states in the partitions. The remaining risk for two parallel test-diagnose-fix tasks is:

$$R_R(G_1, G_2) = \sum_{s \in S_1 \setminus S_o} \Delta_P(s, G_1) P(s) I(s) + \sum_{s \in S_o} \Delta_P(s, G_1) \Delta_P(s, G_2) P(s) I(s) + \sum_{s \in S_2 \setminus S_o} \Delta_P(s, G_2) P(s) I(s)$$

Above, the test sequences of both test-diagnose-fix tasks are denoted by $G_1$ and $G_2$. The maximum remaining risk for a parallel test-diagnose-fix task $i$ is calculated using Equation (16). All possible execution sequences $G_i$ for test-diagnose-fix task $i$ need to be taken into account to determine the maximal remaining risk for a test-diagnose-fix task. The sequence of test cases is fixed, however test cases can pass or fail resulting in a diagnosis of the problem.
and eventually a fix. Therefore, $G_i$ represents all possible pass and fail sequences and this way the maximum remaining risk is calculated.

$$R_{\text{max},i} = \max_{G_i \in \mathcal{G}} \sum_{s \in S_i} \Delta_P(s, G_i) P(s) I(s)$$

The maximal remaining risk of the combined test-diagnose-fix task is the maximum of the maximal remaining risk of the parallel test-diagnose-fix tasks if these parallel test-diagnose-fix tasks are independent. In general, this is not the case and parallel test-diagnose-fix tasks are dependent because of the overlap in fault states. Determining the maximum remaining risk for the combined test-diagnose-fix task requires that all possible combinations of fault states in $\mathcal{P}(S)$ need to be taken into account for both sequences. To reduce the complexity of the calculation the maximal remaining risk for two test-diagnose-fix tasks is estimated using Equation (17).

$$R_{\text{max}} = \max_{G_i \in \mathcal{G}_1, G_j \in \mathcal{G}_2} \left( \sum_{s \in S_1 \setminus S_2} \Delta_P(s, G_i) P(s) I(s) + \sum_{ s \in S_2 \setminus S_1} \Delta_P(s, G_j) \Delta_P(s, G_i) P(s) I(s) \right) + \sum_{s \in S_2} \Delta_P(s, G_j) P(s) I(s) \right)$$

### 5.1 Test stop criteria of the partitioned test-diagnose-fix tasks

The hypergraph partitioning algorithm splits a test-diagnose-fix task into two tasks and combines the cost vector of the two test-diagnose-fix tasks into a combined cost vector. The test stop criterion should still be met after combining the results of the partitioned test-diagnose-fix tasks. Meeting the test stop criterion for (maximal) cost can be assured by setting the stop criteria for the partitioned test-diagnose-fix tasks such that the sum of stop criteria equals the overall stop criterion. The stop criterion for (maximal) duration is obtained in a similar fashion. The stop criterion for duration for the partitioned tasks is equal to the overall stop criterion, because they are executed in parallel. The stop criterion for the remaining risk is more involved, because the remaining risk of the partitioned test-diagnose-fix tasks influences the overall remaining risk, because overlapping fault states are covered in more than one partitioned test-diagnose-fix task. Moreover, the effect of all test-diagnose-fix tasks on these overlapping fault states need to be taken into account to determine if the remaining risk stop criterion is met. The remaining risk of the combined test-diagnose-fix task needs to be determined after every step in the KLFM algorithm, to check if the stop criterion on system level is met using Equations (16) and (17). This is computationally intensive. Therefore, another approach is followed. The remaining risk stop criterion for the parallel test-diagnose-fix tasks is calculated in advance by multiplying the remaining risk stop criterion on system level with a remaining risk factor $\eta_{\text{R}}$. A suitable value for the remaining risk factor needs to be selected for this purpose. A remaining risk factor that is too low leads to less optimal but feasible partitions, while a remaining risk factor that is too high leads to infeasible partitions. The risk reduction in the partitions takes longer than required because a low risk factor is used. This way, the risk stop criterion is reached and feasible solutions are obtained. However, a better solution could be available. A high risk factor on the other hand, results in shorter durations for the partitions, so a better solution could be obtained. However, the combined risk could exceed the risk stop criterion, leading to infeasible solutions.

### 5.2 Estimated objective function

The idea behind the estimator of the test duration, cost and remaining risk is explained first, followed by the definitions. The interleaved test process configuration can be depicted as a directed binary graph. The initial state of this binary graph is the first test in the off-line test sequence. Each test can pass or fail. A passed test case results in the selection of the next test in the sequence. This results in the pass transition from the initial state. The failing test case is the second transition. This way, a binary directed graph is built up. A failing
test case results in a diagnosis and fix task and the next test in the sequence is selected after diagnosis and fixing is completely finished. The directed binary graph of the interleaved test process configuration is depicted in Figure 3. A state is depicted as a circle, a transition as an edge and a leaf node as a triangle. A leaf node indicates that the stop criterion is reached. Multiple leaf nodes can exist, since the stop criterion could be reached via multiple paths through the binary graph. A state, depicted as a circle in Figure 3, is defined as a five-tuple \(X = (p, \Delta p, \varphi, c, r_R)\) with a domain \(\mathbb{R} \times (S \rightarrow \mathbb{R}) \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}\). The elements in the state \(X\) are:

- \(p\), the probability that the state is reached during the test-diagnose-fix task,
- \(\Delta p\), the reduction of the failure probability of each fault state \(s \in S\),
- \(\varphi\), the time passed since the start of the test-diagnose-fix task,
- \(c\), the cost of testing since the start of the test-diagnose-fix task,
- \(r_R\), the remaining risk in the system.

A leaf node is reached if the following condition holds: \((\Phi > \Phi_{\text{target}}) \lor (c > C_{\text{target}}) \lor (r_R < R_{R,\text{target}})\). The leaf nodes of the directed binary graph are used to determine the expected and maximal cost, remaining risk and duration of the combined test-diagnose-fix task. Two transitions are possible from each state in the directed binary graph: a transition if a test passes and a transition if a test fails. The pass transition results in state \(X_p(t, X)\) derived using Equation (18). A fail transition results in state \(X_f(t, X)\) derived using Equation (19).

\[
X_p(t, X) = (p_p(t, \Delta p)p, \delta_{\text{pass}}(t, \Delta p), \varphi + \varphi_{\text{T}}(t), \\
c + C_T(t), r_R(\delta_{\text{pass}}(t, \Delta p))) \tag{18}
\]

\[
X_f(t, X) = ((1 - p_p(t, \Delta p))p, \delta_{\text{fail}}(t, \Delta p), \\
\varphi + \varphi_{\text{T}}(t) + \varphi_{\text{D}}(t) + E(\varphi_{\text{Fix}}), \\
c + C_T(t) + C_D(t) + E(C_{\text{Fix}}), \\
\Delta r_R(\delta_{\text{fail}}(t, \Delta p))) \tag{19}
\]

Both transitions result in a new state according to the pass or fail result of the test case. The transition of the passed test case uses the pass probability \(p_p\) of the test case, while the fail transition uses the fail probability \(1 - p_p\). A different reduction of failure probability \(\delta_{\text{pass}}\) and \(\delta_{\text{fail}}\) is used accordingly. The other difference in the state transitions is the update of the duration and cost with the diagnosis and estimated fix duration for the fail transition due to the interleaved test process. The pass probability of a test \(t\) is defined according to Equation (20). The reduction of the failure probability for all fault states is given as argument \(\delta\) to this function.

\[
p_p(t, \delta) = \prod_{s \in S} (1 - \delta(s)p(s)) \tag{20}
\]
The reduction in failure probability due to a passing test \( \delta_{\text{pass}} \) is described as a function \( \delta_{\text{pass}} : T \times (S \to \mathbb{R}) \to (S \to \mathbb{R}) \) and defined as follows, for \( s \in S \):

\[
\delta_{\text{pass}}(t, \delta)(s) = \delta(s)(1 - R_{\text{pass}}(t, s))
\]

(21)

The remaining risk can be calculated using a certain reduction in failure probability \( \delta \) according to Equation (22).

\[
\Delta R_{\delta}(\delta) = \sum_{s \in S} \delta(s)P(s)I(s)
\]

(22)

The reduction in failure probability due to a failing test \( \delta_{\text{fail}} \) is described as follows. For \( s \in S \):

\[
\delta_{\text{fail}}(t, \delta)(s) = \begin{cases} 
\delta(s) & \text{if } R_{\text{fail}}(t, s) = 0 \\
0 & \text{if } R_{\text{fail}}(t, s) > 0
\end{cases}
\]

(23)

The expected fix duration, \( E(\varphi_{\text{Fix}}) \), is calculated using Equation (24). A number of fixes can be executed in parallel when a diagnosis finishes. The expected fix duration depends on the fix duration of each fault state, as well as the probability that the fix duration is spend. The fix duration of each fault state \( \varphi_{\text{Fix}} \) is known from the system test model. The probability that the fix duration is spent \( p_{\text{Fix}} \) is calculated using Equation (25).

\[
E(\varphi_{\text{Fix}}) = \sum_{s \in S} \varphi_{\text{Fix}}(s) p_{\text{Fix}}(s)
\]

(24)

The idea behind the calculation of the fix duration probability \( p_{\text{Fix}} \) is that the fix duration is spent depending on the probability that the fault state is actually in the system and the probability that the fix duration of another fault state that is to be fixed in parallel takes longer. The probability that a fault state is actually in the system is \( \Delta_{\text{Fix}}(s)P(s) \), while the probability that the fix duration of another fault state takes longer is determined using a sorted list of fault states that are to be fixed. The probability that a fault state is actually in the system is \( \Delta_{\text{Fix}}(s)P(s) \), while the probability that the fix duration of another fault state takes longer is determined using a sorted list of fault states that are to be fixed. A sorted list, \( S_{\text{sorted}} \), is created with the fix duration and fault state as elements. The list is sorted such that the fault state with the highest fix duration is placed first, followed by the fault state with the second highest fix duration and so on. The probability that a fix duration is spend can be calculated for each \( s \in S_{\text{sorted}} \):

\[
p_{\text{Fix}}(s) = \frac{1}{1 - p_{\text{Fix}}(s)} \Delta_{\text{Fix}}(s)P(s) \prod_{d \in S_L} (1 - \Delta_{\text{Fix}}(d)P(d))
\]

(25)

Where, the list \( S_L \) contains the fault states that are already visited. The current element is added to \( S_L \) after \( p_{\text{Fix}}(s) \) is determined.

A single recursive definition for the expected fix duration with signature \( E(\varphi_{\text{Fix}}) : S_{\text{sorted}} \times S \to \mathbb{R} \) that replaces Equation (24) is given in Equation (26).

\[
E(\varphi_{\text{Fix}})(S, S_L) = \begin{cases} 
\varphi_{\text{Fix}}(S) & \text{if } S = \emptyset \\
\prod_{d \in S_L} (1 - \Delta_{\text{Fix}}(d)P(d)) & \text{if } S \neq \emptyset \\
E(\varphi_{\text{Fix}})(\text{tail}(S), S_L \cup \{ S, \emptyset \}) & \text{if } S \neq \emptyset
\end{cases}
\]

(26)

Where \( S, o \) is the first element in the sorted list and \( \text{tail}(S) \) returns the sorted list except the first element of the sorted list, i.e., the remainder of the sorted list or tail. The expected fix cost, \( E(C_{\text{Fix}}) \), is calculated using Equation (27).

\[
E(C_{\text{Fix}}) = \sum_{s \in S} (1 - \Delta_{\text{Fix}}(s)P(s)) C_{\text{Fix}}(s)
\]

(27)

A state in the binary graph represents, among the probability that the state is reached, the time passed since the start of the test-diagnose-fix task and the cost and remaining risk. The end states in the binary graph of nodes represent the states that reached the test stop criterion. The time, cost and remaining risk reached in these end states are used to determine the expected duration, cost and remaining risk for the entire test-diagnose-fix task. The set of all
sequences leading to an end state in the binary graph $G$ is used in Equations (16) and (15). The expected cost and duration of a single test-diagnose-fix task are calculated by taking the average duration and cost of all the sequences in $G$. 

$$\begin{array}{cccccccccccccccccccc}
& s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} & s_{12} & s_{13} & s_{14} & s_{15} & s_{16} & s_{17} & s_{18} & s_{19} & s_{20} & s_{21} & s_{22} & \hline
s_1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_3 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_5 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_6 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_8 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_9 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_{10} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_{11} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_{12} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_{13} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_{14} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_{15} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \hline
s_{16} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \hline
s_{17} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \hline
s_{18} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \hline
s_{19} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \hline
s_{20} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \hline
s_{21} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
s_{22} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \hline
\end{array}
$$

Table 2: System test model (relevant elements) of the example VOIP system

### 6 Example

An extension of the telephone example is used to illustrate the partitioning of system test models. The simple telephone system is extended with a Voice-Over-IP (VOIP) network and another telephone on the other side. So the example system consists of two telephones connected to each other via a VOIP system. The system test model is depicted in Table 2. The impact $I$ of all fault states is set to 1.0. The diagnosis, $x$ duration and apply $x$ duration is set to 0.0 for all fault states and test cases. The test graph of this VOIP system is presented in Figure 4. Two types of nodes can be seen in Figure 4. The circular nodes represent test cases. The grey diamonds represent fault states. The a-priori fault state probability, $P$, is represented by the grey color. A darker tint of grey indicates that the failure probability is higher. The edges in Figure 4 represent the relations, $R_{ts}$, between test cases and fault states. The system test model of the VOIP system consists of 17 test cases with a total test duration of 41
time units. The test graph of the VOIP system is first parallelized with a bin-packing algorithm. The test duration is the only required parameter per test case. A result of bin-packing (bp) are the test sets \( T_{bp}^1 \) and \( T_{bp}^2 \):

\[
T_{bp}^1 = \{4, 5, 6, 7, 9, 12, 13, 14, 17\} \quad (28)
\]

\[
T_{bp}^2 = \{1, 2, 3, 8, 10, 11, 15, 16\} \quad (29)
\]

The test duration of \( T_{bp}^1 \) is the sum of all individual test cases, \( \Phi_{T_{bp}^1} = \sum_{t \in T_{bp}^1} \phi_T(t) \). The test duration of \( T_{bp}^2 \) is determined in a similar fashion. The duration of both test-diagnose-fix tasks executed in parallel is \( \Phi_{bp} = \max(\Phi_{T_{bp}^1}, \Phi_{T_{bp}^2}) \). For the VOIP system, \( \Phi_{T_{bp}^1} = 20 \) and \( \Phi_{T_{bp}^2} = 21 \), so the test duration of both test-diagnose-fix tasks executed in parallel is \( \Phi_{bp} = 21 \).

A graphical overview of the test cases, fault states and relations in the VOIP system is depicted in Figure 5. The resulting test sets \( T_{bp}^1 \) and \( T_{bp}^2 \) are represented by darker and lighter circles respectively. The fault states and relations are presented for completeness purposes. These relations are not taken into account in the bin-packing algorithm. The dotted edges indicate that the connected fault state is covered by only one of the two test partitions. The black solid edges indicate that the connected fault state is covered by tests from two test partitions. This illustrates that the bin-packing algorithm does not take test coverage into account. This graph can be compared with the resulting graph of the test graph partitioning algorithm later on.

![Figure 5: Test graph of the bin-packed VOIP system](image-url)

It can be concluded based on Figure 5, that the bin-packing algorithm does not take the coverage of each test case and the timing related properties of each test and fault state into account, since the selected test cases for each test-diagnose-fix task are scattered. The overlap between the test-diagnose-fix tasks obtained via bin-packing is large, as can be seen by the solid lines in Figure 5. The overlap between the test-diagnose-fix tasks obtained via bin-packing is large, as can be seen by the solid lines in Figure 5.

The adapted HKLFM algorithm does take the coverage and timing properties into account. The step by step adapted HKLFM can be visualized using the VOIP test model of Figure 4. First the test graph is coarsened, i.e., test cases are grouped together. Seven groups are created for the VOIP example: \( H_1, H_2, \ldots, H_7 \). These groups are graphically presented in Figure 6, where it can be seen that test cases are grouped with neighboring test cases. The initial partitioning after coarsening is represented with darker and lighter circles. Initially, one partition consists of one test group with one test case (\( t_{15} \)), while the other partition contains six groups of test cases. Now, groups of test cases are moved to the other partition if this is beneficial for the total cost. The scheme that has been followed in this case is:
Figure 6: Coarsened test graph of the VOIP system

Step 1: Move $t_3$ to partition 2, $\Phi^{HKLFM} = 29.0$.
Step 2: Move test cluster with $t_8, t_9, t_{10}$ to partition 2, $\Phi^{HKLFM} = 13.0$.
Step 3: Move $t_{15}$ to partition 1, $\Phi^{HKLFM} = 12.0$.
Step 4: Move $t_3$ to partition 1, $\Phi^{HKLFM} = 11.0$.
Step 5: Trying the other test groups results in no benefit, so terminate.

Table 3: Partitioning scheme for the VOIP system
Then, the groups of test cases are uncoarsened into single test cases, such that partition $T_1 = \{1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 16, 17\}$ and partition $T_2 = \{8, 9, 10\}$. The same HKLFM procedure is now repeated on the uncoarsened test cases. The resulting test case swaps are:

Step 1: Move $t_{13}$ to partition 2, $\Phi^{HKLFM} = \Pi.o$.
Step 2: Move $t_2$ to partition 2, $\Phi^{HKLFM} = \Pi.o$.
Step 3: Trying the other test cases results in no benefit, so terminate.

Table 4: Uncoarsened partitioning scheme for the VOIP system

Step 2 is beneficial because the number of links between the test-diagnose-fix tasks is lower when $t_2$ is moved to partition 2. The total cost remains the same however.

The final result is depicted in Figure 7. The cost of these parallel test-diagnose-fix tasks is $\Pi.o$. If this result is compared with the initial bin-packing result, then a cost improvement of 47% is obtained for this example. If the resulting cost is compared with the initial (non-parallel) test set with total test cost of 41 time units, then an improvement of 73% is obtained.

Figure 7: Test graph of the VOIP system partitioned by the adopted HKLFM algorithm

7 Case: Lot operations sub-system

In this section, a case study is discussed where the test partitioning algorithm and proposed objective functions are applied. The case study is performed with the lot operations sub-system of an ASML wafer scanner. This sub-system consists of software only. The test-diagnose-fix task that is considered in this case can be executed on a standard SUN Solaris system without requiring additional hardware. Moreover, the fault states in this sub-system are configuration independent, because the same software can be copied and installed on more than one test system. A wafer scanner is used in the critical (lithographic) step in the manufacturing process of integrated circuits (IC's). The manufacturing execution system in an IC factory sends a so-called 'lot' to a wafer scanner. The lot describes the type of job that the wafer scanner needs to perform. These lots are queued in the lot operations (LO) sub-system. The LO sub-system then performs the required setup actions in the wafer scanner and waits until all required material is available to process the lot. The LO sub-system is the main controller of the wafer scanner.

ASML test engineers have defined a system test model of the lot operations sub-system with the following properties:
The test cases that are described in the system test model are developed over a period of a few years. The test cases first developed test the normal operation of LO, while test cases developed in a later stage test additional functionality and special cases where problems were found over the years. The set of 44 test cases, ‘test chapters’, in the system test model contain a few up to hundreds automated test cases. The set of fault states in the system contains fault states for individual components in the system as well as fault states for sub-functions and special high-risk areas. The diagnosis cost and duration is set to 1 hours. This is also the case for the fix cost and duration. We assume here that a test engineer is present when the test cases are executed. Problems found are analyzed and fixed immediately and then testing is continued. This test process matches with the interleaved test process as discussed in Section 5.2.

The test-diagnose-fix task that is partitioned is the final test of a redesign of the LO sub-system. The LO sub-system has been redesigned to accommodate future developments. Additionally, the internal design required an update, because the changes over time in the system resulted in a degradation of the current design. The test-diagnose-fix task is planned to test if the redesigned code still complies with the normal expected behavior of the LO sub-system. This test-diagnose-fix task is parallelized to gain time-to-market. Cost is less important, because enough test resources and developers are available. Three groups of failure probability of the fault states can be distinguished: $P(s) > 80\%$ (4 fault states), $80\% \geq P(s) > 30\%$ (6 fault states) and $P(s) \leq 30\%$ (21 fault states).

7.1 Reference case

First, the duration of the single (non-parallel) test-diagnose-fix task has been determined for reference purposes, because currently all test cases are executed in a weekend, without interleaved diagnosis and fix activities. The current test duration is therefore 56.3 hours. A reference case, with interleaved diagnosis and fixing, needs to be compared with the parallel execution of test-diagnose-fix tasks. The test stop criterion is set to a remaining risk of 1.20, the interleaved test process and an off-line information gain-based test sequencing method are used. No stop criterion on cost and time is defined. The expected duration of the single test-diagnose-fix task is 16.0. The maximal test duration is 37.6 and the expected and maximal cost are 18.18 and 40.13, respectively. The expected remaining risk is 1.08, while the maximal remaining risk is 1.20 according to the target. The test cases are capable of reducing the remaining risk to a level of 1.15, when no faults are present in the system. If faults are present in the system, it depends if the faults are fixed if the remaining risk level of 1.15 is reached. Therefore a slightly higher target remaining risk of 1.2 is chosen, resulting in more feasible sequences.

7.2 Experiments

Several experiments are performed. The performance of the HKLFM algorithm was tested by varying the minimum number of vertices $V_{min}$, the max weight factor $CMaxWghtFrac$ and the number of vertices in a cluster $MaxNrVtxInCluster$. Furthermore, the remaining risk factor $\eta_{\text{R}}$ is varied to determine the optimal remaining risk factor for this case.

The settings for $V_{min}$, $CMaxWghtFrac$ and $MaxNrVtxInCluster$ are summarized in Table 5. The number of vertices in each different coarsening level are summarized under $|V_0|$ for level 0 through $|V_2|$ for level 2. The experiments are marked as C1 through C7. C1 is the experiment where the HKLFM algorithm is used without coarsening and uncoarsening the test-diagnose-fix tasks. The results of the experiments with varying $CMaxWghtFrac$, $MaxNrVtxInCluster$ and $V_{min}$ are presented in Table 6. The number of performed replications is indicated in the # repl. column. The number of replications is 10 for C1 through C3 because the results of these experiments were very close to each other. This way, it is excluded that an accidentally bad or good solution influences the results. Experiments C4 through C7
Table 5: The settings of the performed experiments

<table>
<thead>
<tr>
<th>Exp.</th>
<th>CMaxWghtFrac</th>
<th>MaxNrVtxInCluster</th>
<th>(V_{\text{min}})</th>
<th>(V_0)</th>
<th>(V_1)</th>
<th>(V_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>-</td>
<td>-</td>
<td>(\infty)</td>
<td>45</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>C2</td>
<td>0.01</td>
<td>2</td>
<td>28</td>
<td>45</td>
<td>28</td>
<td>-</td>
</tr>
<tr>
<td>C3</td>
<td>0.5</td>
<td>2</td>
<td>24</td>
<td>45</td>
<td>24</td>
<td>-</td>
</tr>
<tr>
<td>C4</td>
<td>0.1</td>
<td>2</td>
<td>14</td>
<td>45</td>
<td>24</td>
<td>14</td>
</tr>
<tr>
<td>C5</td>
<td>0.006</td>
<td>3</td>
<td>30</td>
<td>45</td>
<td>30</td>
<td>-</td>
</tr>
<tr>
<td>C6</td>
<td>0.12</td>
<td>3</td>
<td>18</td>
<td>45</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>C7</td>
<td>0.025</td>
<td>3</td>
<td>13</td>
<td>45</td>
<td>20</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 6: Results of the different experiments

<table>
<thead>
<tr>
<th>Exp.</th>
<th># repl.</th>
<th>(\mu_\Phi)</th>
<th>(\sigma_\Phi)</th>
<th>(\Phi_{\text{min}})</th>
<th>(\frac{\Phi_{\text{min}} - \Phi_C}{\Phi_C})</th>
<th>CPU [hr]</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>10</td>
<td>12.33</td>
<td>1.31</td>
<td>11.50</td>
<td>0.00</td>
<td>10 \times 15</td>
</tr>
<tr>
<td>C2</td>
<td>10</td>
<td>12.05</td>
<td>0.78</td>
<td>11.66</td>
<td>0.01</td>
<td>10 \times 7</td>
</tr>
<tr>
<td>C3</td>
<td>10</td>
<td>11.79</td>
<td>0.00</td>
<td>11.79</td>
<td>0.02</td>
<td>10 \times 4</td>
</tr>
<tr>
<td>C4</td>
<td>5</td>
<td>14.89</td>
<td>1.18</td>
<td>12.53</td>
<td>0.09</td>
<td>5 \times 4</td>
</tr>
<tr>
<td>C5</td>
<td>5</td>
<td>13.14</td>
<td>1.33</td>
<td>12.05</td>
<td>0.05</td>
<td>5 \times 8</td>
</tr>
<tr>
<td>C6</td>
<td>5</td>
<td>18.89</td>
<td>1.10</td>
<td>16.81</td>
<td>0.46</td>
<td>5 \times 6</td>
</tr>
<tr>
<td>C7</td>
<td>5</td>
<td>16.32</td>
<td>0.00</td>
<td>16.32</td>
<td>0.42</td>
<td>5 \times 6</td>
</tr>
</tbody>
</table>

The remaining risk factor \(\eta_{R\alpha}\) has been varied to determine the optimal remaining risk factor and the effect of varying \(\eta_{R\alpha}\) on the end result. Experiment C3 has been used for this purpose, since experiment C3 delivers the best results in the shortest calculation time. The result of experiment C3 differs 2% from the optimal solution and is determined in 10 \times 4 hours, while experiment C2 differs 1% and is determined in 10 \times 7 hours. The remaining risk factor values used are 0.25, 0.3, 0.325, 0.34, 0.35, 0.36, 0.375. Figure 8 depicts the remaining risk factor versus the minimal test duration. The minimal duration is obtained with a remaining risk
factor of 0.35. Infeasible solutions are obtained when the remaining risk factor exceeds 0.375.

### 7.3 Results

Partitioning the test-diagnose-fix task of the LO sub-system results in a reduction of the expected test duration from 16 hours to 11.50 hours, which is a reduction of around 30%. The maximal test duration is also reduced by around 30%. The expected test cost is increased with roughly 30%, because test cases are executed in parallel. The detailed results are presented in Table 7. The computational effort can be reduced by using coarsening and clustering as set up in experiment C3.

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Expected test duration</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>16.00</td>
<td>11.50</td>
</tr>
<tr>
<td>C3</td>
<td>16.00</td>
<td>11.74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Maximal test duration</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>37.6</td>
<td>25.60</td>
</tr>
<tr>
<td>C3</td>
<td>37.6</td>
<td>25.10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Expected test cost</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>18.18</td>
<td>23.50</td>
</tr>
<tr>
<td>C3</td>
<td>18.18</td>
<td>23.82</td>
</tr>
</tbody>
</table>

Table 7: Detailed results of partitioning the LO test-diagnose-fix task into two parallel tasks. $R_{R,\text{max,target}} = 1.2$ and $\eta_{R_e} = 0.35$

### 8 Conclusions

This paper extends the well-known Hypergraph Kerninghan-Lin and Fiduccia-Matheysses (HKLFM) partitioning algorithm, such that it can be applied to partition test-diagnose-fix tasks. The extension includes a domain specific objective function, where the presence of an edge in a partition influences the outcome of the objective function. The extension of the HKLFM algorithm enables the partitioning of test-diagnose-fix tasks into parallel test-diagnose-fix tasks, while taking the important elements of a test-diagnose-fix task into account: the test sequence, test process, test stop criteria and the system under test itself. An estimator for the objective function has been developed for this purpose and this estimator has been incorporated in the Mondriaan toolset.

The hypergraph partitioning algorithm for test-diagnose-fix tasks has been applied to a case study at ASML. A test-diagnose-fix task of a software sub-system has been partitioned and an improvement in terms of expected test duration of around 30% is obtained. Different parameters of the partitioning algorithm have been varied to investigate the effect of these parameters on the expected test duration and computational effort. An optimal setting for the case study has been determined this way.

A number of possible improvements can be made. The current objective function can be extended with additional test sequencing techniques and test process configurations. This extension broadens the applicability for other types of test-diagnose-fix tasks. Three extensions related to partitioning are proposed. First, the current algorithm supports partitioning a test-diagnose-fix task into two tasks. More than two test-diagnose-fix tasks could be beneficial. This can be done using a recursive algorithm or a flat method algorithm as proposed...
The recursive method can be applied manually using the current toolset. The second extension relates to the partitioning of fault states instead of test cases. The current partitioning method is limited to the partitioning of test cases only, while the partitioning of fault states could also be chosen. The partitioning of fault states should lead to the same results with the current objective functions. It could be possible that more simple estimators of the KPI could be defined if the fault states are partitioned.

The third extension related to partitioning uses the hypergraph partitioning algorithm to partition a large test-diagnose-fix task into two test-diagnose-fix tasks that are executed in series. This way, a rough test sequence, of large groups of test cases, is obtained. These groups of test cases could be sequenced in detail using an optimal test sequencing algorithm [2]. Communication between two parallel test-diagnose-fix tasks is currently not supported, resulting in less optimal results. Communication enables that probability information about fault states present in both test-diagnose-fix tasks is exchanged. Fixing a fault state in one test-diagnose-fix task is then beneficial for the other test-diagnose-fix task(s) also. Communication on the other hand complicates the estimator of the test duration of a single test-diagnose-fix task and could result in an increase of the computational effort.

We would like to thank Kirsten van de Meer (ASML) and Rudi Pendavingh (Eindhoven University of Technology) for their participation in this work.


Authors biography

I.S.M. de Jong has a B.Sc. in Laboratory Informatics and Automation from Breda Polytechnic. He has been a software engineer in various companies in the USA and The Netherlands. Since 1996 he has worked with ASML in systems testing, integration, release and reliability projects. His specialization is in the field of test strategy. Since 2003 he is an active member in the TANGRAM project and a Ph.D. student at the Eindhoven University of Technology. His research concerns integration and test strategies.

R. Boumen received his M.Sc. degree in Mechanical Engineering from the Eindhoven University of Technology, the Netherlands, in 2004. During his work as a master student he worked in the field of supervisory machine control of lithographic machines. Since 2004 he is a Ph.D. student at the Eindhoven University of Technology. His research, embedded in the TANGRAM project, concerns test strategies.

J.M. van de Mortel-Fronczak graduated in computer science at the AGH University of Science and Technology of Cracow, Poland, in 1982. In 1993, she received the Ph.D. degree in computer science from the Eindhoven University of Technology, the Netherlands. Since 1997 she has worked as assistant professor at the Department of Mechanical Engineering, Eindhoven University of Technology. Her research interests include specification, design, analysis and verification of supervisory machine control systems.

J.E. Rooda received the M.S. degree from Wageningen University of Agriculture Engineering and the Ph.D. degree from Twente University of Technology, The Netherlands. Since 1985 he is Professor of (Manufacturing) Systems Engineering at the Department of Mechanical Engineering of Eindhoven University of Technology, The Netherlands. His research fields of interest are modelling and analysis of manufacturing systems. His interest is especially in the control of manufacturing lines and in the supervisory control of manufacturing machines.