Automated Test Mapping and Coverage for Network Topologies

Per Erik Strandberg
Westermo R&D AB, Sweden

Thomas J. Ostrand
Mälardalen University, Sweden

Elaine J. Weyuker
Mälardalen University, Sweden

Daniel Sundmark
Mälardalen University, Sweden

Wasif Afzal
Mälardalen University, Sweden

ABSTRACT

Communication devices such as routers and switches play a critical role in the reliable functioning of embedded system networks. Dozens of such devices may be part of an embedded system network, and they need to be tested in conjunction with various computational elements on actual hardware, in many different configurations that are representative of actual operating networks. An individual physical network topology can be used as the basis for a test system that can execute many test cases, by identifying the part of the physical network topology that corresponds to the configuration required by each individual test case. Given a set of available test systems and a large number of test cases, the problem is to determine for each test case, which of the test systems are suitable for executing the test case, and to provide the mapping that associates the test case elements (the logical network topology) with the appropriate elements of the test system (the physical network topology).

We studied a real industrial environment where this problem was originally handled by a simple software procedure that was very slow in many cases, and also failed to provide thorough coverage of each network's elements. In this paper, we represent both the test systems and the test cases as graphs, and develop a new prototype algorithm that a) determines whether or not a test case can be mapped to a subgraph of the test system, b) rapidly finds mappings that do exist, and c) exercises diverse sets of network nodes when multiple mappings exist for the test case. The prototype has been implemented and applied to over 10,000 combinations of test cases and test systems, and reduced the computation time by a factor of more than 80 from the original procedure. In addition, relative to a meaningful measure of network topology coverage, the mappings achieved an increased level of thoroughness in exercising the elements of each test system.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; **Software testing and debugging**; **Empirical software validation**;

KEYWORDS

testing, test coverage, network topology, subgraph isomorphism

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07.
https://doi.org/10.1145/3213846.3213859

ACM Reference Format:
Per Erik Strandberg, Thomas J. Ostrand, Elaine J. Weyuker, Daniel Sundmark, and Wasif Afzal. 2018. Automated Test Mapping and Coverage for Network Topologies. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213859

1 INTRODUCTION

Communication equipment such as robust routers or switches often provide a communication backbone for critical functions of distributed embedded systems. Such systems range from small local sensor systems to communication and management systems in trains, aircraft or industrial automation. Due to the centrality of some of these systems, software failures in communication equipment can have critical consequences such as delays in data communication, loss of productivity, or even loss of life.

Therefore, software in embedded systems needs to be reliable, and we need to have effective and efficient ways to detect quality shortcomings. It is important to test software and hardware in isolation, as well as testing their integration [1, 12]. One common approach for testing of distributed embedded systems is to build test systems, and to make these test systems available to developers and testers to support manual or automated testing before releasing the software and devices to end users. In order to enable accurate assessment of non-functional behavior (such as timing), it is essential that testing be performed on actual hardware rather than simulated devices.

In the specific case of testing data communication systems, a *test system* provides a configuration of physical devices such as routers and switches connected by a set of links such as various types of cables into a *physical network topology*. For example, a test system may be made up of a switch, three routers and one link breaker, interconnected by a set of DSL, Ethernet and serial cables.

At this level of integration, a *test case* consists of configuration steps, other actions and expected results, arranged in a sequence that exercises a certain feature or functionality of the system under test. Naturally, a test case for a certain feature needs to be executed on a test system that provides the feature to be tested. For example, a test case to verify firewall functionality would require three nodes: the firewall node, an internal node that is to be protected by the firewall, and an external node that attempts to reach the internal node. When the firewall is not activated, the external node should be able to reach the internal node, but after proper activation, the firewall should block this communication. A test framework executing these test cases could perform the required configuration and verification steps.

The specific devices and interconnections required by a test case constitute a *logical network topology* that must correspond to some part of the physical network topology of a test system.

For reasons of efficiency and limited resources, companies typically do not build a specific test system with expensive components for each individual feature being developed. Instead it is common to build a small number of physical test systems, and use each one to test a number of diverse features, by using different test cases.

The central result of this paper is an efficient method to determine, for a given test case, which one of the available test systems can be a platform for execution of the test case. We call this determination the *mapping problem*; for each test case to be executed, we need to *map* the test case's logical network topology onto the corresponding physical network topology of a test system.

Since test cases and test systems can both be represented as graphs, the mapping problem is analogous to determining whether the graph of a test case is isomorphic to a subgraph of a test system. This is an instance of the well-known NP-complete mathematical problem known as the *subgraph isomorphism* problem.[1]

Our work is motivated by a practical problem encountered in an industrial environment that develops communication equipment for distributed embedded systems. Changes are frequent for these products, necessitating frequent modifications to existing test cases, as well as development of new test cases. Integration and automated testing are performed nightly, and each test case must be mapped onto an available test system before it can be executed. An initial solution was a simple software algorithm that performed a search for an appropriate test system to execute each test case, but the search process was inefficient and did not typically generate thorough coverage of the test system structure.

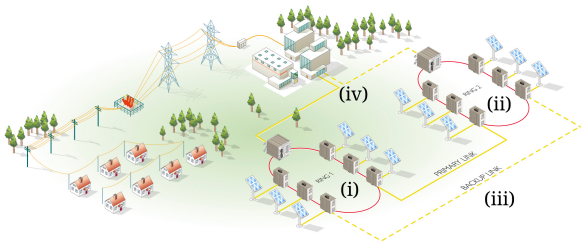
This paper addresses the following issues:

- **Problem 1: Mappings may take excessive amounts of time to determine.** The existing mapping algorithm regularly ran into situations in which test cases could not be mapped onto test systems in a reasonable amount of time. Although the existing algorithm completed the majority of mappings in a few tenths of a second, a substantial number ran much longer, and many cases had to be aborted because of timeouts.
- **Problem 2: Inadequate network topology coverage.** The original mapper did not achieve high coverage in any intuitive sense. A major reason is that the algorithm did not vary the mapping of a test case to a test system. That is, for a given (test case, test system) pair, the algorithm would always choose the same mapping.

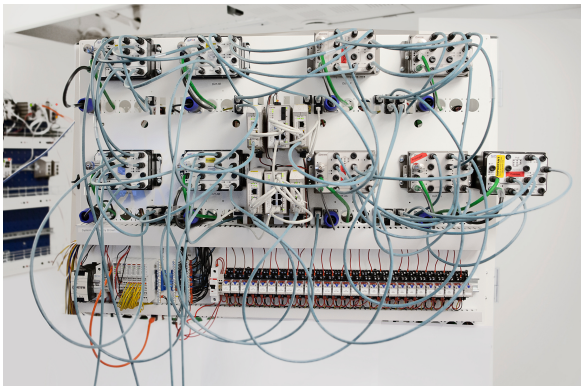
In order to address the above problems, we present a graph-theoretic approach to solving the mapping problem. We then evaluate this approach using industrial data. The results show a substantial decrease in mapping time, an increased network topology coverage, and thus an improved utilization of available test systems.

2 INDUSTRIAL CONTEXT

Westermo Research and Development AB (Westermo) designs hardware and software for robust industrial communication devices. An example of Westermo hardware is robust Ethernet switches while an example of Westermo software is the industrial network operating system – Westermo Operating System (WeOS). A possible scenario for an end customer is to have an installation like the one in Figure 1a, with four principal components: (i) Several devices (or nodes) connected in a ring in the network topology. The ring provides redundancy in case a node goes down, or a cable is lost, perhaps due to an accidental cut during maintenance. In such cases, the other units in the ring can still communicate. (ii) A second ring just like the first. (iii) A primary and a backup link between the rings so that any two nodes in the rings can communicate with each other. (iv) Uplinks to a central facility where an operator can manage the network.



(a) An example of a possible customer installation. This network layout contains many forms of redundancy, which is important for many customers.



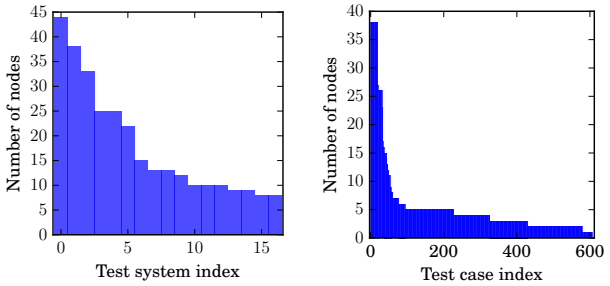
(b) An example of a Westermo test system, built for nightly testing.

Figure 1: In order to test possible customer installation (a), a number of test systems have been built (b).

2.1 Test Systems

To provide rapid feedback for ongoing software development of WeOS, software is tested nightly for regressions using a test framework that has been developed and supported over several years. The nightly testing is performed on a number of test systems. Each test system is set up using a number of nodes (i.e., devices) in a specific network topology. Figure 1b shows an example of one such test system. Over the years, many different test systems have been built at Westermo to test combinations of unique hardware products, unique software protocols, and/or unique customer use cases.

[1]The formal definitions of isomorphism and subgraph isomorphism are given in Section 3.



(a) Size of test systems. **(b) Size of test cases.**

Figure 2: The sizes of test systems and test cases in number of nodes.

In a typical test system, 4 to 25 devices run WeOS. We shall refer to them as devices under test (DUTs). Additionally, the test system may include other nodes with specialized functionality that do not represent devices, but rather allow the user to simulate scenarios that might be encountered in the field. These are known as *link breakers* and are custom-built to be able to simulate scenarios such as cable deterioration by mechanical wear and tear, rust, pests or accidental destruction caused by human factors. Overall, it is typical to have systems ranging in size from seven to dozens of nodes.

The test system in Figure 1b has nine WeOS DUTs, one server, and nodes of other types resulting in about a dozen nodes. Figure 2 shows the wide range of sizes of test systems and test cases studied in this paper. The seventeen test systems studied ranged in size from 7 to 44 nodes. Among a group of 607 test cases, approximately 20 required nearly 40 nodes, the great majority required 5 or fewer nodes, and the smallest tests required only 1 node.

Between a pair of nodes, there may be zero, one or many links (edges). When more than one edge is present, it is possible to test a communication protocol named *link aggregation*, which is popular in applications where redundancy is important. Both the ports on the nodes and the types of cables may vary. For example, some cables may be optical fibers, while others are copper cables. Some ports on some nodes support Ethernet up to a specified speed, while other ports are more limited. The communication protocols supported by different hardware types may also differ. Although physical reconfiguration of the test system is undesirable, this does happen occasionally, for example when a prototype device is replaced with one from production.

2.2 Steps in Nightly Testing

Westermo employs a continuous integration development process, in which new features and changes are developed in separate parallel development branches. Code changes for WeOS are submitted to the source code repository during normal working hours. Each night, the different development branches are automatically built and tested. The overall steps needed for nightly testing are described below and shown in Figure 3:

(1) Using the most recent (successful) commit of each development branch, a new WeOS image to be tested is built for each type of Westermo device.
(2) For each development branch, test cases are selected and prioritized into a branch-specific test suite using the SuiteBuilder tool, described in [8, 9]. Each branch-specific test suite is then assigned an execution slot on a specific test system.
(3) Next, for each test system, and for each test suite associated with that test system, the DUTs are upgraded to the WeOS images under test associated with that suite, and the test cases in the suite are executed one by one according to the following procedure:
   (a) The nodes of the test case are mapped onto a possible subset of the nodes of the test system using the mapper described below.
   (b) The test system devices are reset to a known state.
   (c) The test case is executed.
   (d) The test verdict is reported to a test results database.

2.3 Original Mapping Implementation

The network topology of each test system is written in a configuration file in the YAML language [2], to allow the test framework to understand what nodes there are, and how they are connected. These files represent the *physical* network topology of a test system. The test cases have their requirements (in terms of nodes, edges, models of hardware and so on) described in a similar file that represents the *logical* network topology of a test case.

Prior to executing a test case, a way to map the logical test case network topology onto the physical network topology of a test system must be found. This *mapping* is used to configure the physical network topology to allow it to run the test case. The nodes that are not in use in a certain mapping are configured in such a way that they cannot disturb the execution of a test case: all non-participating ports and communication protocols that were activated by previous test cases are disabled.

The original mapping algorithm has several important shortcomings. It is so slow that when it searches for a mapping for certain test cases without a timeout mechanism in place, it can continue for days, and require human intervention to terminate. Another deficiency of the original mapper is that it always returns the same mapping for a given (test case, test system) pair. Statistics are available that show how frequently a particular node has been used for a certain test, but the mapper does not utilize that information to improve diversity or variability in testing over time.

Since most test cases run in less than a few minutes, it was considered wasteful to spend more than 10 to 20 seconds on finding a mapping before the start of the test case. For slow mappings, a special mapping file can be written to enforce a certain mapping. Creating and maintaining these files require labor-intensive manual work. In addition, updates to the mapping files are not always coordinated with updates of the test systems, which can lead to the testing framework attempting to use a mapping that has invalid information about the latest test systems.

3 PRELIMINARIES

The network topology mapping problem involves the identification of a physical subset of a test system needed to run a logical test

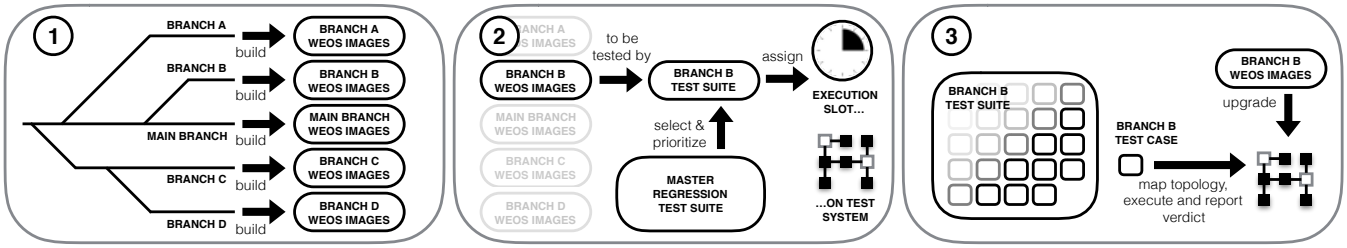


Figure 3: Automated nightly testing procedure.

case on real hardware. In this section we introduce mathematical concepts for discussing and presenting a solution to this problem.

According to standard terminology in graph theory, e.g. [11], a graph is built up of vertices (or nodes), and edges (or links). Two vertices are *neighbors* (or adjacent) if they share at least one common edge. The *degree* of a vertex is the number of neighbors it has. A *walk* is a sequence v_0, e_1, v_1, ..., v_k of graph vertices v_i, and graph edges e_i such that for $1 \leq i \leq k$, the edge e_i has endpoints v_{i-1} and v_i. If there are no repeated edges in a walk, and v_0 is the same as v_k, and no other vertices are repeated, then the walk is a *cycle*. An *even cycle* has an even number of vertices while an *odd cycle* has an odd number of vertices.

Two graphs G and H are *isomorphic* if there is a one-to-one correspondence between the nodes of G and H such that the number of edges joining any two nodes of G equals the number of edges joining the corresponding nodes of H. A graph H is a *subgraph* of a graph G if (i) each of its nodes belongs to the node-set of G: V(G), and (ii) each of the edges in H belongs to the edge-set of G: E(G). The *subgraph isomorphism* problem, for two graphs G and H, is to determine whether G contains a subgraph that is isomorphic to H. For example, in Figure 4, if the nodes of H_1 are associated with the nodes of G_0 as follows: $\{(A,2), (B,4), (C,5), (D,6), (E,7), (F,10)\}$, then the subgraph of G_0 with the nodes {2,4,5,6,7,10} and the edges between those nodes is isomorphic to H_1.

For the physical systems that are being tested in our environment, not only must there be a simple alignment of nodes and edges when identifying subgraphs, there are additional constraints that often come into play, further complicating the algorithm. In particular, some nodes and edges might have attributes that prohibit certain otherwise legal mappings. For example, some nodes may be PCs while others are link-breakers. Similarly some edges might be serial cables while others are Ethernet cables. Therefore, when devising a mapping, not only do we have to consider the identification of a subgraph within a graph, we also have to make sure that the part of the system that we are mapping the test case onto, has the right attributes. In the prototype implementation described in this paper, only the most important node attributes have been implemented. We expect that extending the algorithm to include a broader set of attributes will further improve the performance since the search space will be further limited. We intend to enhance the implementation in this way, and assess the impact on performance.

4 RELATED WORK

The general subgraph isomorphism problem is a difficult, NP-complete problem. Bonnici et al. [3] present an approach for solving the subgraph isomorphism problem using a combinatorial search method. This serves as a starting point for the algorithm presented in this paper. They also discuss related approaches, including the work by Ullmann [10]. While we are unaware of any research that directly addresses the problem we describe in this paper, it is certainly true that graph theoretic concepts have been used in testing-related papers. For example, in early work, Stickney [7] used control flow graphs and cyclomatic trees as test data selection criteria. Similarly, Rapps and Weyuker [6] used graph theoretic ideas in developing their dataflow hierarchy of test data selection and adequacy criteria. In more recent work, Nandi [5] describes the generation of network topologies for use in testing. Another recent paper by Mariani et al. [4] combines graph theory algorithms and machine learning to locate faults in cloud systems. To the best of our knowledge, subgraph isomorphism has not been used to increase network topology coverage in the context of software testing.

5 APPROACH

Let G be the graph in which we want to find one or many instances of the subgraph H. In our industrial setting, G represents a test system, and H represents a test case. We divide the algorithm into four parts:

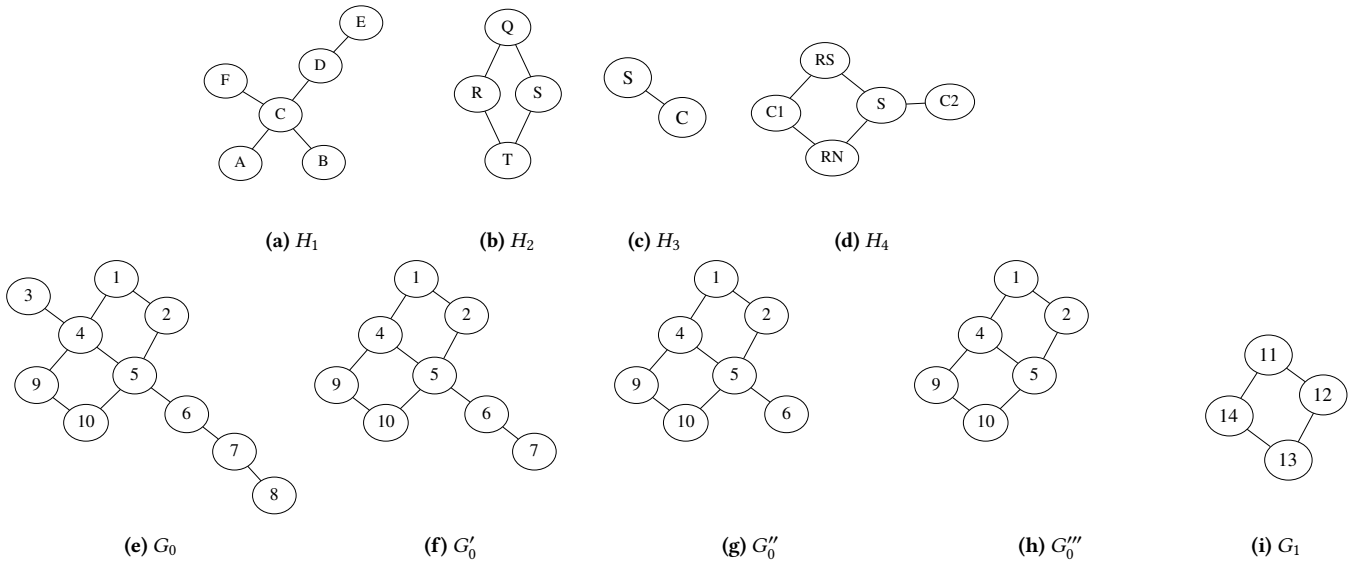
(1) **Preprocessing and Candidates**: identify node candidates and remove unused parts of G.
(2) **Unmappability tests**: stop the search when it can be shown that H cannot be found in G. (In some cases this step also further limits the candidate sets created in step 1).
(3) **Search strategy**: order the candidates as to (i) limit the search space, (ii) rapidly find a match, and/or (iii) find a match while improving network topology coverage.
(4) **Search tree**: build the mapping, one node at a time, by constantly evaluating if what has been built so far is meaningful.

The search process builds a search tree of pairs of nodes from H and G. At the start of the algorithm, the search tree is empty, and if the search completes successfully, one or more mappings $M(H,G)$ can be constructed by traversing the search tree.

Next, we use the graphs in Figure 4 to illustrate the details of the various steps of the search algorithm.

5.1 Preprocessing and Candidates

The following preprocessing steps reduce the size of the target graph G, and create a set of *candidate nodes* for each node in H. These candidates are central to later stages of the algorithm.



(a) H_1 **(b)** H_2 **(c)** H_3 **(d)** H_4

(e) G_0 **(f)** G_0' **(g)** G_0'' **(h)** G_0''' **(i)** G_1

Figure 4: The H graphs represent test cases. G_0 and G_1 represent test systems. The primed versions of G_0 represent successive stages of preprocessing.

It is obvious that a node of H can only be mapped to a node of G with the same or higher degree. This observation is the inspiration for successively removing any node of G that cannot possibly be a candidate for mapping. For example, to map H_2 onto G_0, only nodes of degree 2 or higher are useful. The sequence of steps that successively removes leaf nodes from G_0 yields the graphs G_0' (after removal of 3 and 8), G_0'' (removal of node 7), and finally G_0''' (removal of node 6).

In addition to reducing the candidates based on degree, we also consider constraints on the nodes to limit the number of candidates. As mentioned in Section 3, a PC cannot be mapped to a link-breaker (node constraint), and a serial cable cannot be mapped to an optical fiber cable (edge constraint). In our prototype implementation, we only consider node constraints. In future work we will also include edge constraints.

In some cases, this preprocessing will be sufficient to show that a subgraph H cannot be found in G, such as when H contains a cycle and G does not. The preprocessing algorithm steps are:

(1) For each node z in H, identify all nodes w in G such that z can be mapped to w with respect to degree: $d(z) \leq d(w)$. These are the *candidates* of z.
For example, in Figure 4, when mapping H_1 onto G_0, the candidates of node C are $\{4, 5\}$.
(2) For each node z in H, identify all nodes in the candidate set of z that are not of the same type as z and remove them from the candidate set.
(3) For each node w in G, remove w if it is not a candidate for any node z of H.
(4) If any node was removed from G, repeat from step 1, otherwise stop.

5.2 Unmappability Tests

After creating the reduced version of G, any of the following checks can be used to stop the search as soon as it is determined that no mapping is possible, thereby saving the resources needed to create and parse a large search tree.

5.2.1 ***Counting Nodes and Edges.*** Stop if there are insufficient nodes or edges in G.

5.2.2 ***Degree Sequence.*** The degree sequence of a graph G is the list of the degrees of the nodes of G in descending order. For G_0 in Figure 4e, the degree sequence is $\{4, 4, 2, 2, 2, 2, 2, 2, 1, 1\}$. For H to be mappable to G, it must be the case that the degree sequence of G dominates the degree sequence of H, i.e., when the nodes of both graphs are arranged in degree sequence order, then

$$deg(z_i) \leq deg(w_i), \forall i \in \{1, 2, \ldots, |nodeset(H)|\}$$

For example, if four is the highest node degree in H, and there are five nodes in H with degree four, then there must be at least five nodes in G whose degrees are four or greater.

5.2.3 ***Mapping of Odd Cycles to Odd Cycles.*** Many of the test case and test system graphs include cycles of nodes. If a test case contains a cycle with k nodes, then the test system must also contain at least one cycle with k nodes. The nodes of H that are part of the cycle can only be mapped to nodes of G that are part of such a cycle. This information can further limit the search space by reducing the candidates of nodes in H.

To address this, we use powers of the adjacency matrices of G and H, and walks, in order to find and compare lengths of cycles within the graphs.

The adjacency matrix, $\mathcal{G}$, of a graph G, is a matrix representation of G such that $g_{i,j}$ is equal to the number of edges between nodes i and j of G, which is equivalent to the number of ways to go directly

from node i to node j. Matrix multiplication extends this to longer walks from i to j. Namely, element $g_{i,j}$ of $\mathcal{G}^k$ represents the number of walks of length k from node i to node j.

A non-zero value for element $g_{j,j}$ in the diagonal of $\mathcal{G}^k$ means there is a walk of length k from node j to j. If k is an odd integer, that walk must include a cycle of length at least 3. In addition, because a cycle can be traversed either "clock-wise" or in the opposite direction, the diagonal element will be at least 2. However, if k is even, a non-zero value doesn't necessarily indicate a cycle, because a walk of even length from j to j could simply involve going back and forth between a node and a neighbor.

We use the odd cycle case in order to further decrease the number of possible candidates of the nodes of H:

(1) Call $\mathcal{G}$ the adjacency matrix of G, and $\mathcal{H}$ the adjacency matrix of H.
(2) For each odd power k, $(3, 5, 7, \ldots, n)$, where $n \leq |nodeset(H)|$, compute $\mathcal{G}^k$ and $\mathcal{H}^k$. For practical purposes, we limit $n \leq 21$.
(3) Stop the mapping if there is a larger number of non-zero elements in the diagonal of $\mathcal{H}^k$ than in $\mathcal{G}^k$, as this means that there are more nodes involved in cycles of length k in H than in G.
(4) Update the candidates of a node z in H so that candidates w of G are removed if w is not part of a cycle of length k when z is part of such a cycle.

Using the adjacency matrix in this way, will always correctly eliminate candidates for the shortest cycle of odd length, if such a cycle exists. It may, however, fail to remove candidates for larger values of k, when a small cycle exists.

5.2.4 ***Candidate Depletion.*** For each node z in H, if the candidate set of z is empty, stop the search.

5.3 Search Strategy

Our algorithm uses search trees to try to identify potential mappings, based on an approach described by Bonnici et al. in [3]. Each step in the search is an attempt to map one more node in H to a node in G. A successful search will represent a complete mapping by a path from the root to a leaf that contains one pair for each node in H. Depending on the order in which nodes are assessed for a potential mapping, the search tree may grow larger or smaller. Because neighborhoods of nodes in H must be preserved in an isomorphic subgraph of G, we only consider adding the pair (z,w) to the mapping if the neighbors of w in G include mappings of all the neighbors of z in H. Whenever it becomes impossible to extend a path with a new pair, then that path terminates and we backtrack to the last potential pair in the tree that is still viable. Figure 5c and 5d shows these as dashed hexagonal nodes, such as $(C1, 5)$ and $(C2, 6)$.

Below, two approaches of the search algorithm are presented. Both strategies terminate if no mapping can be found. Strategy 1 finds a mapping if one exists, and then terminates. For Strategy 2, prior mappings can be given as input data, so that this approach reorders the candidates of the nodes in H, and then resumes searching for further mappings. When running Strategy 2 repeatedly *different* mappings are sought. In both strategies, the nodes of H are at first placed in order of the fewest candidates. If two nodes have the same number of candidates, the one of higher degree is placed first. If they have the same degree, they are placed in alphabetical order based on the name of the element they represent.

The idea is to map nodes with few candidates and high degree early, under the assumption that these are hardest to map, and that having few candidates early makes the search space smaller. We illustrate the first approach with an example in section 5.5, and the second approach with another example in section 5.6.

5.3.1 ***Strategy 1: Bonnici-Inspired Search.*** This strategy organizes the search as follows:

(1) Sort the nodes of H in ascending order of the number of their candidate nodes in G. If two nodes have the same number of candidates, then place the node with the higher degree first. If two nodes have the same number of candidates and the same degree, then order them alphabetically by name.
(2) Sort the nodes in each candidate set of G in descending order by degree. If two candidates have the same degree, then order them alphabetically.

5.3.2 ***Strategy 2: History-Aware Search.*** The second strategy organizes the search in a slightly different way:

(1) Order the nodes in H as in Strategy 1.
(2) Order the candidate sets so that, when considering a node z in H to be mapped to a node w in G, place the w with the lowest number of previous mappings of z first. If two nodes have the same number of previous mappings, use the same ordering as in Strategy 1.

5.3.3 ***Consequences of Non-History-Aware Search.*** Consider the graphs for the test case H_3 and test system G_1, in Figures 4c, and 4i. The old mapper and Strategy 1 of the new mapper will always map this test case onto the same nodes, perhaps 11 and 12 with the following mapping: $\{(S, 11), (C, 12)\}$.

However, there are many motivations for a search that favors different mappings: first of all, the DUTs in G are *individuals* – they are not identical and so we want to assure that each type of DUT has been tested. The same is true for the edges of G. One cable might be connected to a port on the same printed circuit board (PCB) as the CPU, while another might be on a separate PCB, far from the CPU. Packets coming in on this port would need to communicate through some chain of internal components in order to reach the CPU, leading to slightly different timing. Even when all DUTs are of the same hardware type and come from the same production batch, it is still desirable to run the test case on different nodes, simply to increase port coverage. However, the DUTs are typically not identical in Westermo's test systems, because a test system is typically built to cover a range of products.

To conclude, using the same mapping is wasteful, since it may fail to trigger issues that might be identified using different mappings representing different devices and connections. We expect that increased network topology coverage will lead to improved fault detection.

5.3.4 ***DUT Coverage – a Type of Network Topology Coverage.*** Clearly network topology coverage is fundamentally different from typical software testing coverage metrics, which generally

aim for coverage of source code lines, branches in if/else-statements, identified risk factors, artifacts such as requirements, etc.

A metric based on covering all possible mappings might at first seem like a suitable network topology metric. However, the number of combinations often grows rapidly to unreasonable numbers, even for a test case of moderate size. For example, we investigated a test case with 4 DUTs and one PC, and counted the number of ways it could be mapped onto the 17 available test systems. We found that for each test system there were between 8 and 1288 unique mappings. Given that this testing is done nightly with a fixed and limited amount of available time, there is generally not enough time to run every test case even once. Consequently, a more pragmatic notion of coverage is required.

In the test systems at Westermo, some nodes have software with very little evolution over time, while other nodes, the DUTs, run the frequently changing WeOS software. This is the rationale behind a metric that measures coverage in terms of the percentage of DUTs used for a (test case, test system) combination. We call this "DUT coverage". Ideally this metric would increase over mappings and reach full coverage after a relatively small number of iterations.

As an example, we could consider the graphs G_1 and H_3 in Figure 4. For a full theoretical coverage we would need eight mappings (there are four ways to place the server node, S, of H in G, and for each mapping of S the client node, C, could either be in a clock-wise, or counter clock-wise position, relative to S). For full DUT coverage, two mappings are needed. One possible mapping places the nodes of H onto nodes 11 and 12 of G; the alternative mapping would use nodes 13 and 14.

The heuristic we use to achieve increased DUT coverage tries to map each node z of H onto every node of G by favoring candidates with the lowest prior coverage of z over candidates with a high prior coverage.

5.4 Search Tree

The search starts with a dummy root node in a search tree, and the nodes of H sorted as described above. The process can be seen as constructing a search tree in which walking from one step to the next means adding a new pair (z,w) of nodes from H and G into the tree. If the search ends with every node of H successfully mapped to a unique node of G, the collection of valid pairs of nodes constitutes a mapping M. If any node(s) of H cannot be mapped, then the search fails.

(1) At the start of the search, all nodes are unmapped, and z_0 is the first node in the ordering of H.
(2) After the nodes z_0, z_1, $\ldots z_{m-1}$ have been mapped, their images in G are w_0, w_1, $\ldots w_{m-1}$, respectively, and the most recent pair in the mapping is (z_{m-1},w_{m-1}).
If there are no unmapped nodes left in H, then the search ends, and the current mapping is a solution (adjacency has already been verified).
(3) Otherwise, call the first unmapped node z_m, and examine whether it has any unmapped candidates. If every candidate of z_m has already been mapped from another node of H, then it is not possible to extend the mapping further from the pair (z_{m-1},w_{m-1}). Mark that pair as dead in the search tree, backtrack up one level in the search tree and try extending from the next sibling pair, at Step 2.
(4) If z_m does have unmapped candidates, call them w_{m1}, w_{m2}, ... In order to determine whether a candidate w_{mj} is a possible mapping of z_m, we have to check whether the mappings of the neighbors of z_m in H are neighbors of the candidate w_{mj} in G.
- For each w_{mj}:
  - for each i in $(1, \ldots, m-1)$: if z_i is a neighbor of z_m, but w_i is not a neighbor of w_{mj}, or if there are fewer links between w_{mj} and w_i (in G) than between z_i and z_m (in H), then add (z_m, w_{mj}) to the search tree, mark it as a dead pair (dashed hexagonal in Figure 5c), and continue to the next candidate of z_m.
  - If (z_m, w_{mj}) is compatible with all previous mapped nodes, then mark the pair (z_m, w_{mj}) as a live pair (ellipses in Figure 5c)
- If z_m has been successfully mapped, let w_m be the first successful target node. Add the pair (z_m, w_m) to the search tree, return to Step 2, and step down into this pair.
- If none of the candidates of z_m can be mapped, then mark (z_{m-1},w_{m-1}) as dead in the search tree, backtrack up one level in the search tree, and try extending from the next sibling pair, at Step 2.
- If all siblings have been marked dead, then backtrack another level towards the root, and return to Step 2.

(5) If all immediate descendants of the root are dead, there is no solution.

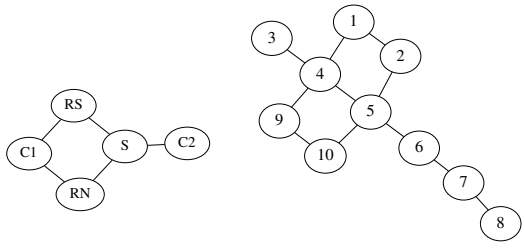
If a valid solution is found in Step 2, then the algorithm may be stopped if one mapping is sufficient. If all possible mappings are sought, then the algorithm continues by backtracking from live leaves of the search tree to the root node.

5.5 Example 1: Bonnici-Inspired Mapping

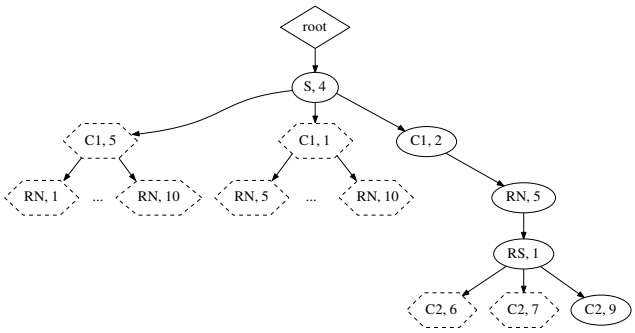
As an illustration of the Bonnici-inspired mapper algorithm, we map the graph H_4 onto G_0, see Figure 5. H_4 could be thought of as a test case for verifying some redundancy protocol in which the designer of the test case wants a server (node S) and two clients ($C1$ and $C2$) to communicate through one of two relays (RS and RN).

Before the first step, the search tree contains only a dummy root, and the nodes from H to be mapped are sorted: {S, $C1$, RN, RS, $C2$}. S is first because it has the greatest degree (largest number of neighbors), and the smallest set of candidates. $C2$ is last because it has the largest number of candidates and lowest degree.

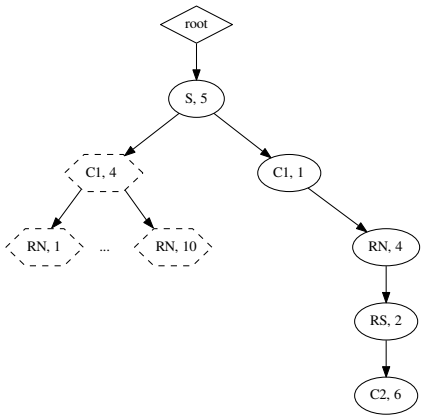
(1) We add (S, 4) to the search tree. We cannot rule this mapping out since there are no neighbors in a graph of only one node.
(2) Now we consider $C1$ with the candidates set of all nodes in G except 3 and 8. After removal of the already mapped 4, a total of seven candidates remain. The first candidate of $C1$ is 5, which is an impossible mapping, but the algorithm does not know this yet. We add the pair ($C1$, 5) to the search tree, and it cannot be ruled out because none of the neighbors of $C1$ in H have been mapped yet. At this point in time the mapped graph only has two isolated nodes S and $C1$.
(3) So far we have mapped S onto 4, and $C1$ onto 5. We next consider RN. The algorithm first tries to map it to 1, but



(a) H_4 **(b)** G_0



(c) Search tree when finding a mapping of H_4 on G_0.



(d) Second iteration of mapping H_4 on G_0.

Figure 5: Test case H_4, test system G_0, and the search tree used to find the first and second mapping of the test case onto the test system.

discovers that its neighbors are not consistent (because *RN* is a neighbor of C in H, but 1 is not a neighbor of 5 in G). The same will be true for 2 and all other remaining candidates of *RN* because $C1$ could never have been mapped to 5 once S is mapped to 4.

(4) We now backtrack from $(C1, 5)$ to $(S, 4)$ and try to map $C1$ onto 1. We discover, again, that *RN* cannot be mapped.
(5) We backtrack again, and try to map $C1$ onto 2; this is consistent, since the mapping so far only has the two isolated nodes S and $C1$.
(6) We again turn to *RN*, and try to map it to 5. This time the neighbors of the mapped nodes are OK, so $(RN, 5)$ is added to the search tree and the algorithm goes forward.
(7) The next node of H is *RS* which, after removing already mapped candidates, has the following candidates: {1, 6, 7, 9 10}. The pair $(RS, 1)$ is OK so we add it to the search tree as a live pair, and move to this pair.
(8) The final unmapped node of H is $C2$, with unmapped candidates: {6, 7, 9, 10, 3, 8}. Trying to map $C2$ to 6 and 7, the neighboring nodes cannot be satisfied, but the pair $(C2, 9)$ is acceptable.
(9) We are now standing in the pair $(C2, 9)$ and have successfully mapped all nodes of H_4 to nodes in G_0. By backtracking from this pair in the search tree to the root, we have a valid mapping.

5.6 Example 2: History-Aware Mapping

We next provide an example of the second search strategy, again mapping H_4 onto G_0, but this time taking the previous mapping into account. In this example we will refer to Figure 5d.

Before the search begins, the nodes of H are sorted in the same way as in the previous approach, but the candidate sets are now different. For example, the candidate set of S has the order $\{5, 4\}$, because S has previously been mapped to 5, but not to 4.

(1) The first pair to be added in this round of the mapping is $(S, 5)$. At this point there are no prohibitions to this pair.
(2) Next we try to add $(C1, 4)$. This will not work, but the algorithm is not aware of this yet.
(3) As in the earlier example, we will not be able to map *RN* to any node of G_0 when $(C1, 4)$ is a pair in the search tree, so we terminate the search and backtrack.
(4) Next we try mapping $C1$ to 1. Again, the graphs in the partial mapping M now contain only isolated nodes, so all neighborhoods are consistent and we can continue.
(5) We try to map *RN* onto 4, and this is OK.
(6) Now we try mapping *RS* onto 2 and this is again OK.
(7) Finally we try to map $C2$ onto 6 and since this is acceptable, we can backtrack from this pair, $(C2, 6)$ and have found another mapping M.

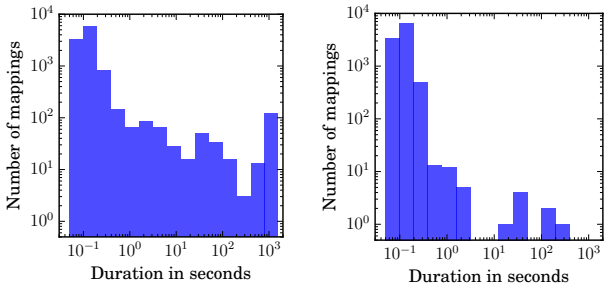
5.7 Implementation

The implementation of the new mapper was coded in two layers: a generic layer of about 700 Python statements for graphs in general, and an additional 200 statements for generating test data. On top of this layer, a Westermo-specific part was added for generating graphs from files in the company-specific YAML format.

The code has four major classes: (i) *Graph*, a class to represent graphs, and export a representation to dot-format for simple compilation into PDF with `graphviz`. (ii) a *Tree* class for handling the search tree, (iii) *Mapper* is the class that implements the search strategies and creates the mappings. (iv) *TestGraphs* is a library for rapidly generating test graphs for running the mapper.

In addition to the standard Python library, the only additional libraries used were `numpy` for generating and working adjacency matrices, and `yaml` for parsing network topology files. Self-tests for the graph and tree class were implemented in `doctest`, and `matplotlib` was used to generate plots for visualization.

An example of an interactive Python session where H_1 is mapped onto G_0 follows:



(a) Old mappings. (b) New mappings.

Figure 6: Distributions of mapping durations. Both plots show the same number of mappings.

```
>>> from network import Graph
>>> from mapper import Mapper
>>> from testdata import TestGraphs
>>> TG = TestGraphs()
>>> G = TG.graphs['G0']
>>> H = TG.graphs['H1']
>>> M = Mapper(G, H, find_all=False)
[...]
>>> (hmap, gmap, _) = M.solutions[0]
>>> print 'H: %s\nG: %s' % (hmap, gmap)
H: ['F', 'E', 'B', 'A', 'D', 'C']
G: [3, 2, 9, 1, 5, 4]
```

6 EXPERIMENTAL EVALUATION

In this section we experimentally evaluate the new mapper in comparison to the old one, using the physical network topologies of 17 test systems and the logical network topologies of 607 test cases. Both the test systems and the test cases are in use at Westermo. In particular, we focus on how the new mapper compares to the old mapper with respect to the two problems stated in the Introduction.

For Problem 1 (Excessive time to determine some mappings), we evaluated the execution time of the old mapper versus the new mapper (Section 6.1). For Problem 2 (Inadequate network topology coverage), we examined the coverage achieved over iterations for the old versus the new mapper (Section 6.2). In both experiments, each of the 607 individual test cases was mapped onto each of the 17 test system topologies. Consequently, we ran more than 10 thousand instances of the old and the new mapping algorithms. If an instance ran for more than 20 minutes (1200 seconds), it was aborted, and considered unmappable.

6.1 Problem 1: Slow Mappings

Both the old mapper and the new mapper are implemented in Python. However, all measurements were done outside of Python, using a bash script. Thus, the measured time duration includes not only the time to search for the mapping, but also the time needed for starting the Python interpreter, parsing files of the topology descriptions for both the test case and test system, and allocating data structures for the graphs. This overhead is about 80 to 100 ms for each (test case, test system) pair. In practice, both the old and the new mapper required only 100 ms for a very large number of mappings. These mappings were not considered a primary concern for Westermo. The new approach is rather aimed at reducing the time needed for the minority of mappings that required the longest amount of time.

Table 1: Durations of mappings for old and new approaches, in seconds.

Mapper	total	min	max	mean	median	std.dev.
Old	161099	0.05	1200	15.6	0.11	129.5
New	1928	0.07	242	0.19	0.10	3.0

Because of the excessive time required for these mappings, the total mapping time with the old implementation for the 10,319 combinations of test cases and test systems was about 44.7 hours. The new implementation took 33.1 minutes to complete the same 10,319 combinations, representing a roughly 84-fold total speedup. On average, the old implementation required 2.6 hours to map the 607 test cases onto each test system, while the new implementation averages 1.9 minutes per test system. With the much higher speed of the new implementation, many additional test cases can be mapped and run on each test system every night. Figure 6 illustrates the distributions of the durations achieved with the different mapper implementations, using a log scale.

The minimum and median mapping durations were about the same for both the old and the new method: less than .07 seconds for the minimum and less than 0.11 seconds for the median. The old method accomplished 95.5% (or 9860) of the mappings in less than 1 second, while with the new method the corresponding number was 99.8% (or 10302). Thus we see that almost all test cases can be mapped onto test systems very quickly using either approach.

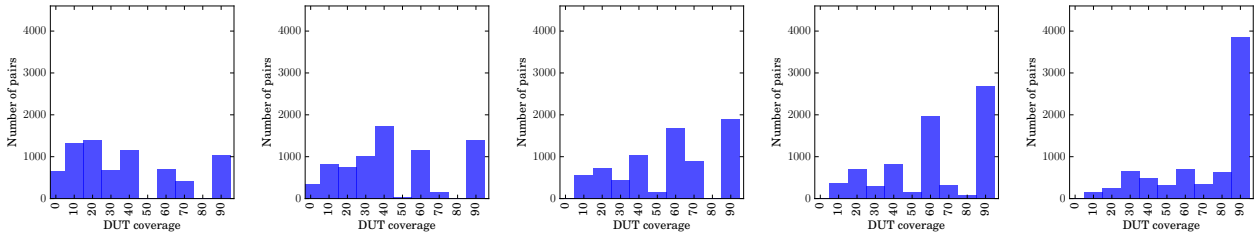
However, using the old implementation, the maximum duration was the timeout of 20 minutes, and 117 mappings required more than 1000 seconds. With the new implementation, only 3 mappings required more than 100 seconds, and the maximum duration was 242 seconds. It was therefore never necessary to abort a test case mapping because it took too long using the new mapper. This result is significant because those long-running test cases sometimes prevented the complete running of the entire test suite overnight. In fact, this was a major part of the motivation for seeking a new mapper implementation.

The *total, minimum, maximum, mean, median* and *standard deviation* of all mappings are shown in Table 1.

We envision that refining the new mapper software to handle all the requirements of all test cases would impose a minor penalty on *all* test cases, but would likely improve performance in the worst cases, in particular when requirements reduce the number of candidates, and also the size of the search tree.

6.2 Problem 2: Inadequate Coverage

Because the old mapper always identifies the same mapping for a given (test case, test system) pair, it can miss many potential mappings of a test case. In Section 5.3 we proposed a history-aware search strategy to make H "move around" over G to increase the test coverage of the network topology. In order to evaluate this,



(a) 1 iteration. (b) 2 iterations. (c) 3 iterations. (d) 4 iterations. (e) 5 iterations.

Figure 7: Distribution of DUT coverage after from one to five iterations, using the new history-aware search strategy.

we repeatedly ran the history-aware version of the new mapper for each (test case, test system) pair, Unmappable pairs, and test cases that do not include any DUTs at all were excluded from the results, since coverage, as defined in this paper, is not meaningful in those cases. After excluding these pairs, more than 7400 pairs of test cases and test systems were used to evaluate DUT coverage.

Figure 7 illustrates the increase of DUT coverage as the number of iterations increases, with the x-axis representing decile coverage. Figure 7a shows the distribution after one iteration. At this point, the median (test case, test system) pair achieves 33% DUT coverage. Note that for the single iteration case, the old mapper implementation would get the same result, as would the non-history-aware version of the new mapper. Figures 7b, 7c, 7d, and 7e, show the improving coverage distribution (shifting towards 100% DUT coverage) as the number of iterations increases for the history-aware mapper. After only 5 iterations, the median pair of test case and test system is at 100% DUT coverage.

In practical terms, if we assume each test case is executed once per test system each night, and if the history-aware mapper keeps track of mappings from previous days, the median DUT coverage would gradually increase towards 100%. In the example of Figure 7, 100% is achieved after 5 days; for other cases it would take less or more time, depending on the number of test cases and complexity of the test systems. Empirical studies with additional systems could be carried out to learn likely rates of achieving high coverage, and to investigate how those rates are affected by the size and complexity of test systems and test cases. In general, the new mapper should produce improvements in coverage compared to the old implementation, and a consequent increase in fault detection.

7 DISCUSSION AND CONCLUSIONS

In this paper we have described and evaluated a new method for improving test coverage of embedded communication devices in network topologies. The new method is significantly faster than the old method, and by taking historic mappings into account, test coverage increases rapidly over time.

Manual intervention is needed to cope with the shortcomings of the old method. The speed of the new method should significantly reduce the need for manual work.

Both of our mapping strategies look for solutions to the subgraph isomorphism problem, and are modifications of the search tree process presented by Bonnici et al. [3]. The approach in [3] is essentially a breadth-first attempt to build all possible mappings that discards partial mappings that cannot be completed. Our approach, in contrast, is depth-first in the sense that it constructs a mapping as far as possible, backtracking to the first place another alternative is feasible when a partial mapping cannot be completed. Strategy 1 attempts to find a single mapping from test case to test system, while Strategy 2 is a novel extension that attempts to find previously unused mappings for better network coverage. Additional innovations of our approach are the preprocessing and unmappability steps that reduce the search space before applying the subgraph matching process. In particular, reducing the test system graphs, mapping cycles onto cycles, and the history aware search strategy are all new features. Furthermore, our implementation was developed from scratch without access to any previous code.

Bonnici et al. originally developed their algorithm to be used for bioinformatics applications, and we have modified the algorithm and described its usage in a different environment. It is clear that with at most minor modifications, our algorithm could be adapted for other domains that rely on a network structure, including transportation and financial networks.

The new mapper is a prototype implementation and some aspects of the filtering process for candidates could be further improved. We expect that in a future implementation, this will impose a small penalty in the performance of all mappings, cause no significant change in the general case, and yield a significant improvement in cases for which improved candidate reduction would lead to a reduced search tree. We expect that this will further decrease the amount of manual intervention necessary as compared to the use of the older method.

The approach we propose can, of course, be improved, and future work will include: (i) taking all attributes on nodes and edges into account, (ii) work on candidate reduction for cycles of even length, [13], and (iii) comparisons of performance when using approaches other than combinatorial search. However, as shown by our experimental assessment, even the prototype version can make a measurable difference in the speed and comprehensiveness of the testing process in practice.

ACKNOWLEDGMENTS

This work was supported by Westermo Research and Development AB, the Swedish Knowledge Foundation (grants 20130085, 20130258, 20150277 and 20160139) and the Swedish Research Council (grant 621-2014-4925).

REFERENCES

[1] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2016. On Testing Embedded Software. *Advances in Computers* 101 (2016), 121–153.

[2] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. 2005. YAML Ain't Markup Language (YAML™) Version 1.1. *yaml. org, Tech. Rep* (2005).

[3] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14, 7 (2013), S13.

[4] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. 2018. Localizing Faults in Cloud Systems. In *Int. Conf. on Software Testing, Validation and Verification (ICST)*. IEEE.

[5] Sunit Kumar Nandi. 2016. Topology generators for software defined network testing. In *Int. Conf. on Electrical, Electronics, and Optimization Techniques (ICEEOT)*. IEEE, 2984–2989.

[6] Sandra Rapps and Elaine J. Weyuker. 1985. Selecting software test data using data flow information. *IEEE transactions on software engineering* 4 (1985), 367–375.

[7] Mike E Stickney. 1978. An application of graph theory to software test data selection. In *ACM SIGSOFT Software Engineering Notes*, Vol. 3. ACM, 111–115.

[8] Per Erik Strandberg, Wasif Afzal, Thomas Ostrand, Elaine Weyuker, and Daniel Sundmark. 2017. Automated System Level Regression Test Prioritization in a Nutshell. *IEEE Software 2017* 34, 1 (April 2017), 1–10.

[9] Per Erik Strandberg, Daniel Sundmark, Wasif Afzal, Thomas J Ostrand, and Elaine J Weyuker. 2016. Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors. In *Int. Symp. on Software Reliability Engineering (ISSRE)*. IEEE.

[10] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42.

[11] Robin J. Wilson. 2010. *Introduction to Graph Theory, 5th Edition.* Prentice-Hall.

[12] Wayne H Wolf. 1994. Hardware-software co-design of embedded systems. *Proc. IEEE* 82, 7 (1994), 967–989.

[13] Raphael Yuster and Uri Zwick. 1997. Finding even cycles even faster. *SIAM Journal on Discrete Mathematics* 10, 2 (1997), 209–222.