Faster Exact Reliability Computation

Vincent Débieux, Yvonne-Anne Pignolet, Thanikesavan Sivanthi
ABB Research, Switzerland, firstname.lastname@ch.abb.com

Abstract—**High reliability guarantees are a prerequisite for any critical infrastructure. In complex systems, the computation of the probability to provide a service is difficult and hence time consuming. To this end, this article presents an improved method for exact reliability computation by exploiting the existence of articulation points in graph models for such systems. Evaluated on generalized block graphs with many articulation points, this method provides a speed-up of 50% to 85% in general. When the network does not contain articulation points, the overhead in memory and computation time is negligible in comparison to an existing method.**

I. Introduction

With the advent of the Internet of Things (IoT), the number of components in industrial automation systems will continue to grow. To benefit from this trend in critical infrastructure such as the electricity distribution grid and industrial control networks, the specific requirements with respect to dependability guarantees need to be taken into account. As the systems will become increasingly inter-connected, the analysis of dependability aspects becomes more and more challenging.

One key aspect of dependability is reliability, *the ability of an item to perform a required function, under given environmental and operational conditions and for a stated period of time* [1]. The functions considered in automation systems depend on computing devices as well as multiple input and output devices (I/O nodes). These devices are interconnected with each other via networking components, i.e. network devices and links. As a consequence, a subset of all these devices (nodes) and links need to be operational at all times for the correct operation of the system function. The components of interest for a specific function can be described by a boolean expression, where the variables correspond to the operating state of the components ('`true`' if they are working, '`false`' otherwise). Computing the reliability then consists in calculating the probability that the boolean expression evaluates to '`true`'. Depending on the complexity of the automation networks and functions this task can be very challenging. In general it is an NP-hard problem [2].

Having a fast reliability computation method for automation network functions allows us to assess if a system satisfies its reliability requirements. In case it does not then one can determine the "bottlenecks" of the systems using importance measures as proposed by Rausand and Høyland [1] which are based on the repeated execution of reliability computations. Thus an efficient engineering process requires fast reliability computation methods.

In this article, we focus on a subtle optimization of exact reliability computation. More specifically, we study an improvement of an efficient algorithm, proposed by Lin et al. [3] and adapted to network functions by Sivanthi and Görlitz [4]. Note that it is straight-forward to apply this method to the computation of the availability of automation network function.

II. Background and Related Work

Reliability computation algorithms typically transform the network topology and the dependency function into a graph, enumerate all mincuts/minpaths on this graph and then compute the probability that the vertices of the graph are working for all mincuts. More precisely, the vertices of this graph are the computing and I/O devices and also the networking components. A cut is a set of vertices that make the function fail if they are all down. It is minimal (mincut) if no subset is also a cut. A path is a chain of nodes and links that connect a source s to a target t. It is minimal (minpath) if no subset is also a path. There are two main issues when computing the reliability of a system. First, minpath/mincut enumeration is a NP-hard problem [2]. Second, minpaths/mincuts are not disjoint which transforms the reliability computation minpaths/mincuts into a large and complex probability calculation [1].

Several exact reliability computation methods have been presented over the years. The very first methods used Sum of Disjoint Product (SDP) [5]–[7] which is impractical for medium sized networks because of the exponential number of minpaths/mincuts produced. Then appeared factoring/decomposition methods [8], [9] that are more efficient but still unfeasible for medium sized networks. An equivalent efficiency is provided by the Inclusion/Exclusion methods [10]. Currently, the most efficient methods use *Binary Decision Diagrams* (BDDs) [3], [11] for the efficient storage and manipulation of large boolean expressions.

Lin et al. [3] propose an efficient *recursive merge* mincut algorithm for computing the reliability and availability of networks. This method was later adapted to the evaluation of functions by Sivanthi and Görlitz [4]. The advantage of this method is that it enumerates the mincuts using the recursive merge algorithm on $s-t$ graphs (graphs with special source/target nodes s and t) constructed using communication paths between the I/O nodes and the computing nodes (called *dependent nodes*). This prevents the recursive merge algorithm from loosing computation time on irrelevant parts of the network. After the mincut computation, a BDD is used to recursively compute the reliability of the function. The current implementation in Sivanthi and Görlitz [4] suffers from long computation times for large network sizes. We improve on these methods by exploiting the existence of *articulation points* [12] in the $s-t$ graphs constructed from typical automation network topologies.

III. Method

As discussed above, the reliability computation method for network functions [4] applies the recursive merge algorithm [3] on $s-t$ graphs. Each of these graphs in turn represents part of the dependency function and a subset of communication paths derived from the network topology. How to compute these $s-t$ graphs is described in detail in [4].

While the automation network might be highly connected to offer redundancy, the $s-t$ graphs constructed as the input to the recursive merge algorithm often contain articulation points, i.e., nodes that disconnect the graph if removed. Splitting the graph at articulation points creates biconnected components (in which we have at least two vertex-disjoint paths between every pair of nodes). By definition, articulation points are cuts and even mincuts since they are cuts of size 1. We can exploit the existence of articulation points in the $s-t$ graphs by splitting the graph at all the articulation points and running the recursive merge algorithm separately on the biconnected components. This can reduce the time to compute the mincuts of the original graph drastically, while computing all correct mincuts because a mincut can only involve nodes belong to the same biconnected component. Otherwise, either it is simply not a cut, or it is not minimal.

IV. Evaluation

A. Setting

The networks we study satisfy the following assumptions:

- The topology is connected.
- Component failures are independent with a constant rate.
- Nodes and links are either working or in a failed state.
- Communication paths between I/O nodes and the dependent node are of minimum length, precomputed off-line.

The networks used in the first part of this evaluation are created using a random network generator. The generated networks resemble typical automation network topologies. In particular, the biconnected components of the underlying network graphs are cycles, series, cliques or complete bipartite graphs (generalized block graphs, cf Pignolet et al. [13]). Thus the generated networks include combinations of trees, ring networks (HSR [14]) and parallel networks (PRP [14]). In some of the experiments, random edges are added to the generated topology to test the limits of the method when the networks become more complex and have fewer articulation points. When adding random edges, all pairs of nodes are considered and an edge is added between the node pairs with some probability.

The dependent node and the I/O nodes of the dependency function are chosen uniformly at random among all nodes. In most of the experiments there are only *AND* operators present in the dependency function, i.e., there is no redundancy in the input and output nodes, but there can be redundant communication paths. Some experiments also use *OR* operators in the dependency function. In order to keep the networks analyzable, these *OR* operators involve at most four input or output nodes. In other words, *OR* operators are only allowed in the lower levels of the dependency function's parse tree.

Box-and-Whisker plots are used to represent the data resulting from the following experiments. The box goes from the 1st quartile to the 3rd quartile. The median (2nd quartile) is represented by an horizontal line inside the box. The whiskers length is set to 1.5 times the inter-quartile range (width of the box). Outliers are represented by points below and above the whiskers. Unless otherwise stated the default parameters of the experiments are a network size of 100 nodes, dependency functions using only *AND* operators, 30% of the nodes serve as input or output nodes, and a maximum of 4 paths is considered between each input or output node and the dependent node. In all experiments, measurements are gathered from 100 network instances that are generated with different seeds. All experiments are conducted on a Linux machine with 3.2 GHz Intel CPU and 8GB RAM.

B. Random Networks

The proposed method is evaluated on random networks with a generalized block graph structure, which captures the connectivity of current as well as future industrial automation systems, which will be interconnected with other systems over an industrial internet comprising multiple networks and topologies [15] to enable advanced analytics and optimized operations. The legends in the plots correspond to the following:

boost-rm The original recursive merge algorithm implemented using the Boost Graph Library (BGL)[1].

boost-ap Splits the graph at articulation points before using *boost-rm* on each biconnected component.

In the following we compare average computation times. Note that also on a per-network basis boost-ap never takes longer than boost-rm to compute the minimum cuts.

1) Network size: Figure 1 shows how these methods scale with regard to the number of nodes in the network for different proportions of I/O nodes. The ratio of nodes that are articulation points in the networks ranges between 10% and 90%, with an average of around 50%. Accordingly the number of articulation points in the $s-t$ graphs analyzed for each dependency function can also vary a lot. For the case with 10% of I/O nodes, there are some outliers which are one order of magnitude above the 3rd quartile. This is due to the randomness in the choice of the I/O nodes. When the number of I/O nodes is small, the I/O nodes can end up all close to the dependent node or all far from it. However, this is less probable when the number of I/O nodes increases. This can be clearly seen for the case for 50% of I/O nodes. Note that *boost-ap* is on average 81% more efficient than the original recursive merge algorithm.

2) Maximum number of paths: Figure 2 shows how the maximum number of shortest paths between each I/O node and the dependent node affects the computation time of the recursive merge algorithm. *Boost-ap* is not influenced by the number of paths because the $s-t$ graphs evaluated here contain several articulation points, allowing it to split the network

[1]http://www.boost.org/doc/libs/1_63_0/libs/graph/doc/

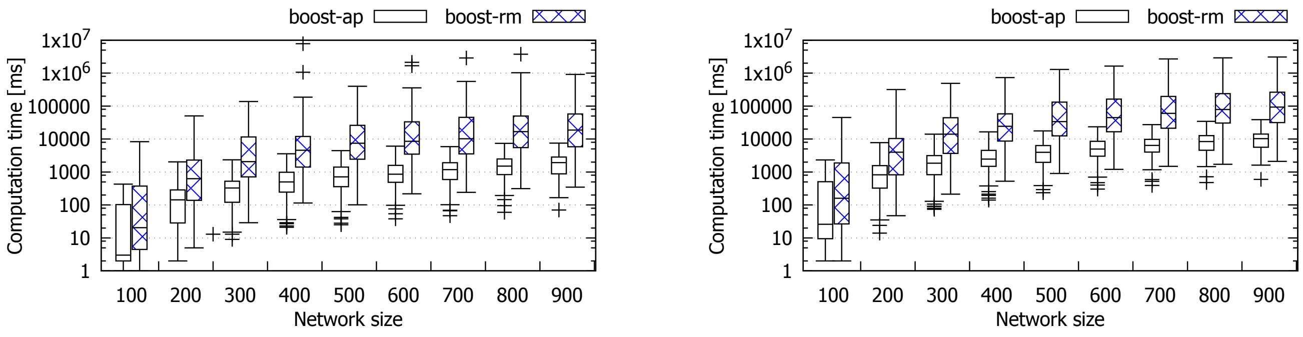

Figure 1. Computation time w.r.t. a) *network size* using 10% of I/O nodes, b) *network size* using 50% of I/O nodes

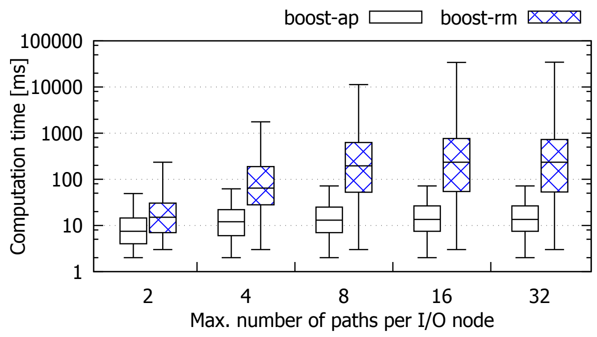

Figure 2. Computation time w.r.t. *maximum number of paths* between each I/O node - dependent node pair

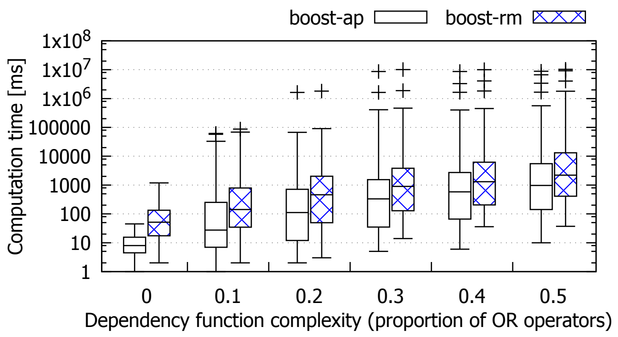

Figure 3. Computation time w.r.t. proportion of *OR* operators on networks of 80 nodes

into biconnected components. For the original recursive merge algorithm, the computation time increases exponentially as the number of paths varies from 2 to 8, but it then becomes constant when the number of paths is further increased. This is because there are only a few I/O nodes that have more than 8 paths to the dependent node. Thus the computation is very similar for 8, 16 and 32 paths. When the number of paths is equal to two, the $s-t$ graphs are simple and *boost-ap* on an average performs 47% better than *boost-rm* but with 4 paths and more, it reduces the computation time by 85% on average.

3) Dependency function complexity: Figure 3 shows how the complexity of the dependency function affects the computation time on networks with 80 nodes. The complexity of the function is configured through the proportion of *OR* operators out of all operators. Even with that limitation the growth in computation time is close to exponential. This is expected because *OR* operators imply more complex $s-t$ graphs with fewer articulation points for the recursive merge algorithm. The articulation point method still provides an improvement of around 52% in computation time for this case.

4) Random edges: This experiment tries to see how both methods scale if we break the generalized block graph structure of the networks by adding random edges. Doing this decreases the number of articulation points, reducing the effectiveness of *boost-ap* compared to *boost-rm* and increasing the complexity of $s-t$ graphs. Figure 4 shows exactly that behavior. Already for a small probability of random edges (2%) the computation time explodes and the performance of *boost-ap* is similar to that of *boost-rm*. The average ratio of articulation points in the network topology is around 50% without random edges and drops down to 7% for networks with 2% random edges and further decreases roughly inversely proportionally to higher random edge probabilities. Interestingly, adding more random edges does not increase the computation time beyond a certain limit: the paths between I/O nodes and the dependent nodes become shorter, but they also become more and more disjoint. This means shorter but wider $s-t$ graphs. When getting closer to a clique network, all $s-t$ graphs become similar, and thus the standard deviation in computation time gets smaller.

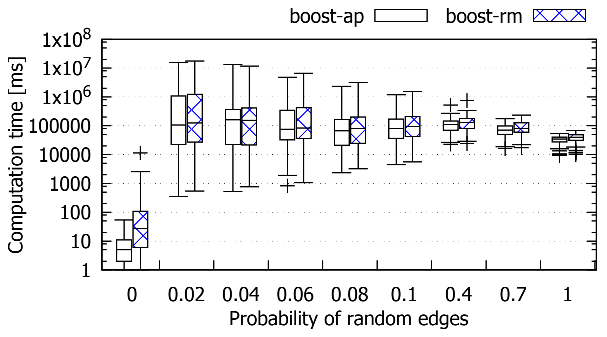

Figure 4. Computation time w.r.t. *probabilities of random edges* in networks of 60 nodes and 8 paths.

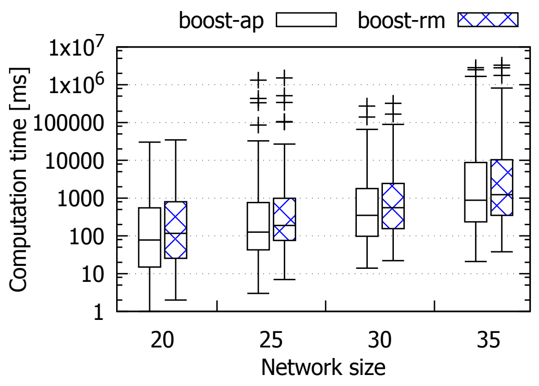

Figure 5. Computation time w.r.t. *network size* with 3% of random edges, 30% of I/O nodes, 20% of *OR* operators and 8 paths

5) Putting things together: It is interesting to see the effect on the computation time of our method when all these parameters are considered together. Figure 5 shows the running times for 3% random edges, 30% of I/O nodes, 10% of *OR* operators and a maximum of 6 paths between each I/O nodes and the dependent node. For a network of size 35, more than 75% of the networks are analyzed in less than 10 seconds, but in some cases, the computation times goes up to more than 20 minutes. Such complex networks contain fewer articulation but the proposed method still performs 32% better than the original version.

C. Substation Automation Networks

Finally, we evaluate our method on typical substation automation network topologies for 10 to 60 bays using the HSR and PRP protocols. The reliability computation times for the different networks presented in [16] are shown in Table I. These networks have very few articulation points, and the number of devices varies between 10 and 80. The dependency function corresponds to the conjunction of the devices of all bays as I/O nodes. The improvement thanks to the exploitation of articulation points is considerable even for such networks. The average speed up using the proposed method on the networks is around 17.5% on average.

Table I
COMPARISON OF ARTICULATION POINTS METHOD AND ORIGINAL METHOD ON SUBSTATION AUTOMATION NETWORKS

network	boost-ap (in ms)	boost-rm (in ms)
hsr10	8	49
hsr40	958	1141
hsr60	3544	3931
hybrid-hp10	2	2
hybrid-hp40	42	60
hybrid-hp60	74	92
hybrid-ph10	66	86
hybrid-ph40	295	359
hybrid-ph60	403	523
prp10	2	2
prp20	7	9
prp30	12	12
prp40	18	18
prp60	30	30

V. Conclusion

This article evaluates an improvement method for the reliability computation of industrial automation networks. In particular, the existence of of articulation points in such networks' graph representations is exploited to speed up the reliability computation. Experiments show that the method performs better and reduces the computation time by 50% to 85% on average in comparison to the approach that simply performs a recursive merge for enumeration of mincuts on random networks with many articulation points. Further experiments on real-world industrial automation networks show that the improved method also performs considerably better than the original method on highly redundant networks with few articulation points.

For further work, it would be interesting to apply this method on the approximation approach described by Sebastio et al. [17], to offer a trade-off between accuracy and time complexity.

References

[1] M. Rausand and A. Hoyland, *System Reliability Theory: Models, Statistical Methods, and Applications*. JOHN WILEY & SONS INC, 2003. [Online]. Available: http://www.ebook.de/de/product/3608074/marvin_rausand_arnljot_h_yland_system_reliability_theory_models_statistical_methods_and_applications.html

[2] M. O. Ball, "Computational complexity of network reliability analysis: An overview," *IEEE Trans. on Reliability*, vol. 35, no. 3, 1986.

[3] H.-Y. Lin, S.-Y. Kuo, and F.-M. Yeh, "Minimal cutset enumeration and network reliability evaluation by recursive merge and BDD," in *Proc. Eighth IEEE Symp. Computers and Communications. ISCC*, 2003.

[4] T. Sivanthi and O. Grlitz, "A systematic approach for calculating reliability of substation automation system functions," in *Proc. IEEE Int. Energy Conf. and Exhibition (ENERGYCON)*, Sep. 2012, pp. 957–962.

[5] S. Hariri and C. S. Raghavendra, "Syrel: A symbolic reliability algorithm based on path and cutset methods," *IEEE Trans. on Computers*, vol. C-36, no. 10, pp. 1224–1232, Oct. 1987.

[6] W.-C. Yeh, "A new sum-of-disjoint-products technique to determine network reliabilities with known minimal paths," in *Proc. Third Int. Conf. Information Technology and Applications (ICITA)*, 2005.

[7] W. C. Yeh, "An improved sum-of-disjoint-products technique for symbolic multi-state flow network reliability," *IEEE Trans. on Reliability*, vol. 64, no. 4, pp. 1185–1193, Dec. 2015.

[8] K. B. Misra, "An algorithm for the reliability evaluation of redundant networks," *IEEE Trans. on Reliability*, vol. 19, no. 4, 1970.

[9] R. K. Wood, "Factoring algorithms for computing k-terminal network reliability," *IEEE Trans. on Reliability*, vol. 35, no. 3, 1986.

[10] Y. H. Kim, K. E. Case, and P. M. Ghare, "A method for computing complex system reliability," *IEEE Trans. on Reliability*, vol. 21, no. 4, 1972.

[11] Y.-R. Chang, H.-Y. Lin, I.-Y. Chen, and S. yen Kuo, "A cut-based algorithm for reliability analysis of terminal-pair network using obdd," in *Proc. 27th Computer Software and Applications Conf. COMPAC*, 2003.

[12] R. Tarjan, "Depth first search and linear graph algorithms," 1972.

[13] Y. A. Pignolet, S. Schmid, and G. Tredan, "Adversarial Topology Discovery in Network Virtualization Environments: a Threat for ISPs?" *Distributed Computing*, vol. 28, no. 2, pp. 91–109, 2015.

[14] IEC, "Industrial Communication Networks High Availability Automation Networks Part 3: Parallel Redundancy Protocol (PRP) and High-availability Seamless Redundancy (HSR)," 2010.

[15] Industrial Internet Consortium, "The Industrial Internet of Things Volume G5: Connectivity Framework," https://www.iiconsortium.org/IICF.htm, 2017.

[16] L. Thrybom, T. Sivanthi, and Y. A. Pignolet, "Performance analysis of process bus communication in a central synchrocheck application," in *Proc. IEEE 20th Conf. Emerging Technologies Factory Automation (ETFA)*, Sep. 2015, pp. 1–9.

[17] S. Sebastio, K. S. Trivedi, D. Wang, and X. Yin, "Fast computation of bounds for two-terminal network reliability," *European Journal of Operational Research*, vol. 238, no. 3, pp. 810–823, 2014.