# PyMBD: A Library of MBD Algorithms and a Light-weight Evaluation Platform

**Thomas Quaritsch and Ingo Pill**
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/II, 8010 Graz, Austria,
{quaritsch, ipill}@ist.tugraz.at

## Abstract

Facing a diagnostic challenge, the decision about the algorithmic variant to adopt for a certain project is a challenging one. With PyMBD, we provide an easily accessible Python library, offering our implementations of several established model-based diagnosis (MBD) algorithms that a user can experiment with. Via the integrated experimentation features, a user can also delve into assessing the effects of her algorithmic changes and optimizations. That is, algorithm authors can run evaluations and generate comparison graphs via accessing simple interfaces to algorithms and backends such as SAT solvers. C-based algorithms (or algorithm parts) can be interfaced via the Boost.Python library.

## 1 Introduction

That the identification of possible root causes for encountered issues is not a trivial task at all has been known for a long time, and motivated a lot of research in the Artificial Intelligence community about providing effective solutions to tackle this problem (see e.g. [1; 2; 3; 4; 5; 6; 7]). However, when facing such a diagnostic challenge, there is the immediate and natural question of which algorithmic variant to actually implement. While papers like [8] can help us in assessing some options, they provide experimental data only for specific experimental domains chosen by the authors. A library offering a selection of algorithmic variants that one can conduct own experiments with, could certainly help in easing such a decision. It might also help MBD adoption for practical purposes in general, by offering a means to assess MBD performance in general without the need to practically implement the algorithms. Last but not least, such a library might also be helpful in assessing our own new ideas for research in the context of diagnosis, if it is easily accessible, suitable for rapid prototyping, and offers an effective interface for doing quick (or extensive) experiments with algorithmic variants.

In this paper, we present our library PyMBD that we have been developing and using in recent years in our research in this very context (e.g. [8; 9; 10; 11; 12; 13]). PyMBD was implemented in Python [14], in order to support rapid prototyping of new ideas using the convenient data types and programming style of Python as well as its available scientific libraries like SciPy [15]. Via the Boost.Python library [16], one can also interface C++ code, so that based on profiling information, one can investigate performance bottlenecks and explore corresponding optimizations when implementing performance-critical algorithm parts in lower-level C++ code, like we did for [9]. As discussed in Section 2, PyMBD offers effective options for evaluating an algorithm's run-time and memory consumption in the context of included or self-written experiments. In terms of benchmark samples (see Section 3.1), we provide a fault injection engine for the well established ISCAS'85 digital circuits, as well as samples that we obtained during our research on behavioral diagnosis of specifications in the Linear Temporal Logic (LTL) [12; 17]. Please let us note that, besides offering a selection of MBD algorithms, the library also allows to experiment with some Minimal Hitting Set (MHS) algorithms in isolation (see Section 3), an algorithmic problem that is central to conflict-based consistency-oriented model-based diagnosis, in that diagnoses can be computed as the MHSs of the set of encountered conflicts.

Complementing full-fledged evaluation frameworks like the DX-Competition framework [18], our aim was to provide an easily accessible library that comes with several well-established MBD algorithms (see Section 3) and allows for experimenting with those variants as well as newly derived fresh ideas. Our focus was on the use of simple interfaces and the user experiencing a steep learning curve, in order to make MBD easily accessible.

## 2 Framework Architecture

Our framework[1] mainly consists of a Python package providing the diagnosis algorithms, (SAT) solver interfaces, benchmark functionality, and some C++ code that allows a user to integrate C++ algorithms via a CPython module and the Boost.Python library. After extracting the package, the user will find the following directory structure:

- **pymbd** is the main Python package containing the framework, algorithms and code for benchmark scenarios, with the following sub-packages:
  - **algorithm**: diagnosis algorithm implementations
  - **benchmark**: scenario-specific code and utility functions for the benchmarks
  - **diagnosis**: main interface for executing diagnosis algorithms in different configurations
  - **sat**: interface code to solver backends
  - **util**: helper functions

[1]http://modiaforted.ist.tugraz.at/downloads/pymbd.zip

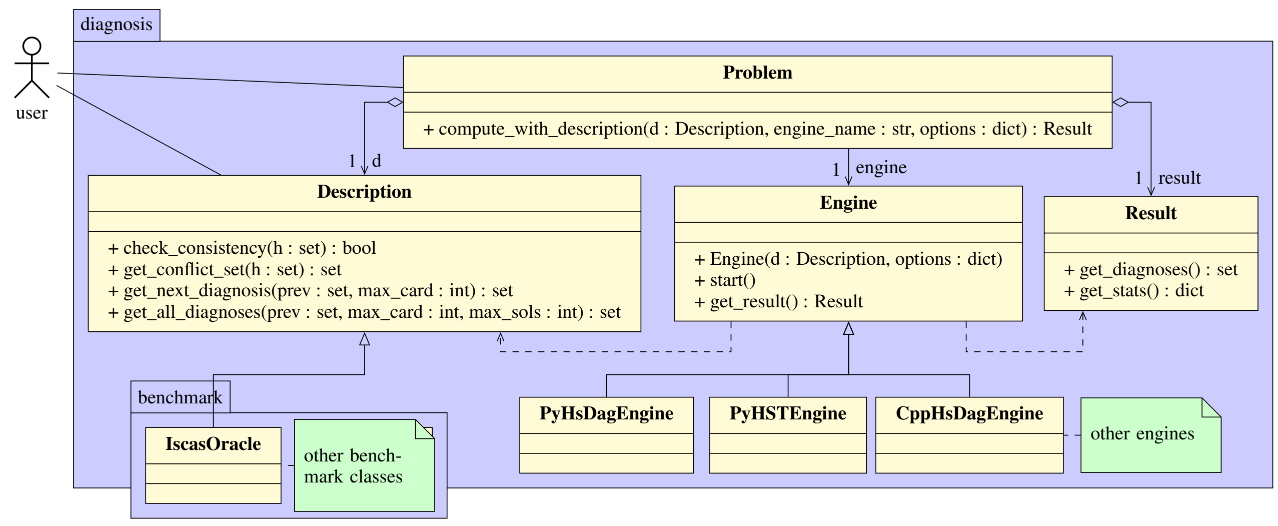
diagnosis
user
Problem
+ compute_with_description(d : Description, engine_name : str, options : dict) : Result
1 d
1 engine
1 result
Description
+ check_consistency(h : set) : bool
+ get_conflict_set(h : set) : set
+ get_next_diagnosis(prev : set, max_card : int) : set
+ get_all_diagnoses(prev : set, max_card : int, max_sols : int) : set
Engine
+ Engine(d : Description, options : dict)
+ start()
+ get_result() : Result
Result
+ get_diagnoses() : set
+ get_stats() : dict
benchmark
IscasOracle
other bench-mark classes
PyHsDagEngine
PyHSTEngine
CppHsDagEngine
other engines

Figure 1: UML class diagram of the diagnosis interface

- **benchmarks**: scenario data and results, scenario-specific execution/evaluation code
- **cresources**: CPython module containing interface code to the Python part as well as C++ diagnosis algorithms (created using Boost.Python)
- **lib**: contains the solver backend sources/executables

In order to provide a clean, object-oriented way for accessing the diagnosis algorithms (from a user's point of view) and support an easy implementation of new algorithms, we defined a clean interface, parts of which are depicted in the UML class diagram in Figure 1.

Figure 2 shows a short example on how to use this interface for computing diagnoses using the smallest ISCAS85 circuit *c17* (see Section 3.1 for a report on included scenarios). Line 4 shows how to set up a scenario-specific **Description** object called **IscasOracle**, which is constructed from the circuit description itself—the (converted[2]) ISCAS85 netlist—and an observation (Boolean values for the inputs and outputs of the circuit, in order of their appearance in the netlist file). Line 5 instantiates a diagnosis **Problem** object, whose **compute_with_description** method is then called with the description, a diagnosis engine handler name and a configuration **max_card=2** (i.e., computing up to double fault diagnoses) in Line 6. The **Result** object **r** is used to retrieve computed diagnoses and run-time statistics.

When implementing a new algorithm, we have to subclass **Engine** and register a corresponding (and new) handler name in **pymbd.diagnosis.engine.ENGINES**. Similarly, in case we want to add a new scenario this requires us to subclass the **Description** class and implement its methods as seen in Figure 1 - for example by means of the SAT interfaces in the corresponding **sat** sub-package. When choosing C++ for (parts of) the implementation, the algorithm must be equipped with a Python interface (via Boost.Python) in order to be accessible by the framework. For this purpose, the interface code to access Python-based **Description** objects is already in place, converting method arguments and return values correspondingly.

[2]The ISCAS85 package contains a C converter from the original netlist files with extension `.isc` to a simpler format denoted with extension `.sisc` here.

```
1  from pymbd.diagnosis.problem import Problem
2  from pymbd.benchmark.iscas.oracle import IscasOracle
3  from pymbd.util.sethelper import write_sets
4  o = IscasOracle('c17.sisc', [0, 1, 1, 1, 1], [1, 1])
5  p = Problem()
6  r = p.compute_with_description(o, 'hsdag-yices',
       max_card=2)
7  d = r.get_diagnoses()
8  d = map(o.numbers_to_gates, d)
9  print write_sets(d)
10 # [{10gat,23gat},{22gat,19gat},{11gat},
11 #  {16gat},{10gat,19gat},{22gat,23gat}]
12 print r.get_stats()['total_time']
13 # 0.0206670761108
```

Figure 2: Usage Example

The **benchmarks** directory contains input and output data of evaluation runs. In PyMBD, we group our runs into *experiments*, where every experiment is kept in a subdirectory and may contain multiple *runs* (i.e., with different parameters or algorithm variants). Experimental data is kept in tab-separated text files. This ensures that these data can be easily viewed and edited manually with every text editor, while supporting also easy access for an automated processing using arbitrary programming languages or via importing the data into spreadsheets. A file named **problems.txt** contains one line for each diagnosis sample per experiment, and can store meta-information about the sample such as the problem size. The actual input data is stored in a corresponding line in **inputs.txt**. A Python script named **compute.py** can be used to parse these files and run a set of algorithms on these inputs. For each run, a sub-directory is created that contains single (tab-separated) **results-*.txt** and **output-*.txt** files per algorithm, again using one line for each sample. The statistical data gathered about each sample in the results file can be customized for each experiment (via **data_structures.py**). Using **plot_graphs.py** the results such as run-time and memory usage can be further processed to create graphs using gnuplot [19]. The **compute.py** script maintains a status file **status.txt** containing information about the currently computed sample, such that the computation can be interrupted and resumed (continuing with the next unfinished sample) at any time. Each sample is computed in a new, clean process such that executions do not interfere with each other. A second process uses the operating system's **ps** utility[3] to record the maximum resident set size (RSS) of the computation process and the selected solver (see Section 3.1 for a list of supported backends). For timing information we focus on user-experienced run-time, that is, wall clock time determined by Python's **time.time()**. In order to get consistent timing results for detailed evaluations, we suggest the user to disable the GUI and swapping, so that interruptions to the process are limited.

## 3 A Library for Model-Based Diagnosis

For an introduction to the topic of consistency-oriented model-based diagnosis, we refer the reader to [4]. In the following, and due to space restrictions, we assume that the reader is familiar with the principle of model-based diagnosis and its aim of identifying possible root causes for encountered conflicts between a system's experienced and expected behavior (in the context of a given system description). As, e.g. Reiter does in [2], we use $COMP$ do denote the set of components, $OBS$ to refer to the observations, and $SD$ as synonym for the system description that contains some assumption predicates $AB(c)$ about whether a component $c$ operates abnormally or not.

In the following, we briefly depict the algorithm implementations and benchmarks already coming with PyMBD.

Our framework contains four algorithms: two conflict-based ones (HS-DAG and HST) as well as two variants computing diagnosis directly using a SAT solver. That is, for each algorithm, we do offer several implementation variants (including, for instance, some C-Code, or using different backend interface variants – see Section 3.1). HS-DAG is Greiner et al.'s corrected version [20] of Reiter's original diagnosis algorithm [2], and HST is Wotawa's variant [21] of Reiter's idea. Those algorithms are based on an implicitly defined list of conflicts, which is dynamically calculated by a theorem prover (SAT solver) as needed. The direct ones on the other hand do not require the computation of conflicts, but instead query a solver for diagnoses in the abnormal predicates directly. To this end, Direct-MaxSAT constructs a MaxSAT problem, while Direct-SAT introduces cardinality constraints in a pure SAT search approach.

**HS-DAG [2]:** Reiter's original algorithm maintains a tree, where each non-leaf node $n$ is labeled with a conflict $C$. For each component $c$ in this conflict, there is an outgoing edge $e$ labeled with $c$. Node labels $C$ are chosen such that the set of edge labels $h(n)$ on the path from the root to $n$ may not intersect with the node's label. New conflict sets are retrieved by checking a new node $n$'s set of edge labels $h(n)$ for consistency, assuming $h(n)$ to be a diagnosis, i.e., its components to operate abnormal (all others normal). When consistent, $n$ is a leaf and the edge-labels $h(n)$ represent a diagnosis. Otherwise, a new conflict set is derived as label for this node. The tree is constructed in a breadth-first manner, and several ideas are used to prune the tree and ensure the minimality of the derived solutions. [20] presented with HS-DAG an improved version that uses a directed acyclic graph and addresses some minor but serious flaws in Reiter's original formulations.

Wotawa's variant **HST** [21] tries to avoid constructing nodes that would be pruned by HS-DAG. Adopting an idea used for subset computation, HST orders the elements in $COMP$, and, defined by those elements encountered in the tree so far (with a focus on the current path), the algorithm limits a node's outgoing edges to some range in $COMP$. Thus, it omits some edges HS-DAG would construct, but only when the corresponding label combinations are investigated in other branches anyway (if they are of interest).

Both algorithms can use an internal set $CC$ of conflicts that is extended whenever a new conflict $C$ is encountered. Thus $CC$ acts as cache to SAT-solver calls, where we first search in $CC$ for some $C$ not intersecting with $h(n)$. A cache miss followed by a satisfying SAT instance then identifies a leaf.

HST and HS-DAG can be limited efficiently to compute only diagnoses up to a given cardinality by stopping node expansion at the corresponding DAG/tree level. The backbone of the HS-DAG implementation is the compact **python-graph** library (version 1.8), which is built upon DAG-global neighbor and incidence hash maps (Python **dict**s). In addition, we keep reverse hash mappings from node labels and potential hitting sets ($h(n)$) to their corresponding nodes for an efficient implementation of the node reuse and pruning rules. A list of leaves (grouped by cardinality) speeds up involved subset checks. For the implementation of HST, we replaced the graph library with a single node class as HST builds a tree structure only. A HST node forms a tree using a mapping *node label* $\rightarrow$ *child node* for its children and a parent node pointer.

**Direct-MaxSAT** (see also [8]) is our first implementation of a direct approach. Based on the idea of MERIDIAN [5], we construct a MaxSAT problem as a set of clause/weight pairs. While clauses forming $SD$ are assigned weight $\infty$ (i.e., amounting to a partial MaxSAT problem), each assumption $\neg AB(c)$ is added with weight 1. The MaxSAT solver is queried for solutions until the maximal desired cardinality has been exceeded (or the solver returns UNSAT). In each step, a blocking clause for the previous solution is added, avoiding it and its supersets as subsequent solutions.

For this approach, we use the Yices SMT solver [22] and peruse its extended assertions to state a (partial) MaxSAT problem: (cf. our Yices ISCAS85 model in Section 3.1)

$$\forall c \in COMP: \texttt{(assert+ } \neg AB_c \texttt{ 1)}$$

Using Yices' **(max-sat)** command, we obtain a maximum satisfiable subset in terms of abnormal predicates. The blocking clause for a diagnosis $\Delta$ is added via **(assert** $\neg\Delta$**)** for the following computation.

**Direct-SAT** (see also [8]) is based on cardinality constraints (networks) as proposed in, for example, [23; 7]. By adding a limit on the number of abnormal predicates activated simultaneously, a satisfying solution of

$$SD \wedge OBS \wedge (AB_1 + AB_2 + \cdots + AB_{|COMP|} \leq k)$$

results in a diagnosis of cardinality $k$ at most. Incrementing $k$ from 1 to the maximal desired cardinality while blocking discovered solutions (and their supersets) like in the MaxSAT case, we obtain a pure SAT diagnosis algorithm.

A classic way to encode cardinality constraints into the Boolean domain is the use of sorting networks, e.g., Odd-Even Mergesort (OEMS) networks [24]. The idea is to transform the summands $AB_i$ into a unary number (i.e. "sort" all the 1s to the left) and then add the clause $\neg x_{k+1}$, where $\{x_i | 1 \leq i \leq |COMP|\}$ are the sorted bits. As it is often the case that $|COMP| \gg k$, we use an encoding tailored

[3]Therefore, the script currently supports Linux and OS X only.

to this situation, coined Cardinality Networks [25]. Compared to OEMS networks, they require only $\mathcal{O}(n \log^2 k)$ clauses instead of $\mathcal{O}(n \log^2 n)$, where $n$ is the number of inputs (i.e., $|COMP|$ in our case). Like [7] we use SCrypto-Minisat, a SAT solver capable of returning all (or multiple) solutions to a given instance. Thus unlike Direct-MaxSAT, with Direct-SAT we obtain all diagnoses for a specific cardinality by a single query to the reasoning engine.

**Minimal Hitting Set Computation** With minimal hitting set computation being an important topic in the context of MBD (in that diagnoses can be computed as minimal sitting sets of encountered conflicts), PyMBD contains several corresponding algorithms to experiment with, such as those we used in [9; 10; 11]. This includes several algorithms besides HS-DAG and HST, i.e., algorithms that do not, or better, cannot compute the set of conflicts on-the-fly, but rather depend on them being pre-computed. PyMBD contains implementations of corresponding algorithms such as the Boolean algorithm [26] or STACCATO [27], as well as SAT and MaxSAT-based MHS algorithms similar to Direct-MaxSAT and Direct-SAT.

## 3.1 Available Scenarios and Backends

**ISCAS85** Our framework contains code to create scenarios based on the combinational circuits from the well-known ISCAS85 benchmark suite [28]. We also provide a test case generator that allows to create diagnosis problem for the ISCAS85 circuits by introducing an arbitrary number of faults into a circuit. The generator iteratively mutates gates in the circuit, while after each mutation the influenced (changed) outputs are frozen for the remaining iterations, such that the individual faults do not mask each other entirely. Diagnosis problems for the ISCAS85 circuits can be constructed in Yices [22] syntax, as well as conjunctive normal form (CNF) to be tackled by SAT solvers like PicoSAT or SCryptoMinisat (see below).

**Yices Model:** Besides being an SMT solver which could be of interest for specific problems, Yices has two very interesting features that motivated us to provide a corresponding interface in PyMBD. First, it can be launched in daemon mode in order to solve multiple similar problems by retracting old and adding new assertions (so that not all "learned" data is lost between the instances). Second, its conflict search can be focused on our $AB(c)$ predicates, using an extended type of assertions. Our corresponding Yices model for the ISCAS problems is as follows,

$$\begin{aligned}
\forall (v,b) \in \mathit{OBS} &: \texttt{(define } v\texttt{::bool } b\texttt{)} \\
\forall c \in \mathit{COMP} &: \texttt{(define } AB_c\texttt{::bool),} \\
&\quad \texttt{(assert } \neg AB_c \rightarrow \mathit{out}_c := f_c(\mathit{in}_c^1, \ldots)\texttt{)} \\
\forall c \in \mathit{COMP} \setminus h &: \texttt{(assert+ } \neg AB_c\texttt{)} \\
\forall c \in h &: \texttt{(assert+ } AB_c\texttt{),}
\end{aligned}$$

where $h$ is the set of edge labels $h(n)$, and $\mathit{out}_c$, $\mathit{in}_c^i$ and $f_c$ are a component $c$'s output-/input signals and its Boolean function implemented using Yices' built-in Boolean operators. In subsequent calls, only $AB$ predicate assignments are updated while the system description is retained. A satisfiable instance indicates a consistent assignment $h$, for an unsatisfiable one, we derive an unsat core in terms of $AB(c)$ predicates as new conflict.

In this context, our Yices backend provides different interface variants that can be exploited by HS-DAG and HST. The first one, coined *combined interface*, uses a single method, intended to compute a new conflict, whose empty result denotes consistency of a set $\Delta$. The *split interface* variant separates those concerns using two methods, possibly even distributed to two Yices instances. It aims to exploit the fact that a solver may reuse obtained intermediate information (such as learned clauses) to speed up subsequent calls if only small parts of the problem have changed.

**Dimacs Model:** Many SAT solvers, including PicoSAT and SCryptoMinisat, use the so-called DIMACS input format. It encodes a CNF, such that variables are represented by integers (with negative ones representing negated variables) and clauses are space-separated integer lists terminated by (the reserved) "0".

**PicoSAT** is a popular SAT solver written by A. Biere [29]. PicoSAT uses files for input and output, such that in our experiments we placed the corresponding directory in a RAM drive, such that the execution times are not influenced by hard disk accesses. PicoSAT can compute unsatisfiable cores for a given problem, allowing us to compute new conflicts. We filter the unsat-core output file keeping only those clauses assigning assumption predicates ($\neg AB_i$) to form a conflict to be used by HS-DAG or HST.

A. Metodi's **SCryptoMinisat** [30] is an extended version of CryptoMiniSat [31]. While it does not support the computation of unsat cores, it allows us to compute multiple (or all) solutions to a given SAT problem using only one call. As our experiments showed, this makes it especially suitable for direct SAT-based diagnosis approaches. Rather than adding blocking clauses for found solutions externally (in the diagnosis algorithm), they are added in an internal loop in the solver, resulting in far better run-times. It is, however, important for diagnosis applications that these blocking clauses can be limited to a certain range of variables (our $AB$ predicates), as otherwise we would compute countless redundant solutions.

**LTL Diagnosis** For our work on the diagnosis of formal temporal descriptions written in the Linear Temporal Logic, we also included our encoding of LTL into a CNF and the corresponding scenario code to our framework. For details we refer the reader to the corresponding papers [12; 13].

## 4 Summary

In this paper, we described our publicly available PyMBD library that offers several consistency-oriented model-based diagnosis (and minimal hitting set) algorithms that a user can experiment with. For some well-established algorithms, we offer several implementation variants that might, for instance, incorporate C-Code for performance-critical parts or slightly different backend interfaces (see the discussion of our Yices model in Section 3.1). Our options for conducting run-time and memory consumption experiments (for included or self-written scenarios) allow a user to explore performance effects for both, provided and self-developed algorithms. That is, via our choice of Python and our simple diagnosis interface (see Figure 1), PyMBD supports the user in rapid prototyping of own ideas. Summarizing, with PyMBD we aimed at providing an easily accessible MBD library with a light-weight experimentation platform, that anyone can access, experiencing a steep learning curve.

## References

[1] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24(1-3):347–410, 1984.

[2] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[3] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[4] W. Hamscher, L. Console, and J. de Kleer, editors. *Readings in Model-based Diagnosis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[5] A. Feldman, G. Provan, J. de Kleer, S. Robert, and A. van Gemund. Solving model-based diagnosis problems with Max-SAT solvers and vice versa. In *Proceedings of the 21th International Workshop on Principles of Diagnosis*, 2010.

[6] J. de Kleer. Hitting set algorithms for model-based diagnosis. In *22nd International Workshop on Principles of Diagnosis*, pages 100–105. 2011.

[7] A. Metodi, R. Stern, M. Kalech, and M. Codish. Compiling model-based diagnosis to Boolean satisfaction. In *AAAI Conference on Artificial Intelligence*, pages 793–799, 2012.

[8] I. Nica, I. Pill, T. Quaritsch, and F. Wotawa. The route to success - a performance comparison of diagnosis algorithms. In *International Joint Conference on Artificial Intelligence*, pages 1039–1045, 2013.

[9] I. Pill, T. Quaritsch, and F. Wotawa. From conflicts to diagnoses: An empirical evaluation of minimal hitting set algorithms. In *22nd Internationl Workshop on Principles of Diagnosis*, pages 203–210, 2011.

[10] I. Pill and T. Quaritsch. Optimizations for the Boolean approach to computing minimal hitting sets. In *20th European Conference on Artificial Intelligence*, pages 648–653, 2012.

[11] I. Pill and T. Quaritsch. And yet another variant of Reiter's complete on-the-fly hitting set algorithm. In *24th International Workshop on Principles of Diagnosis*, pages 210–215, 2013.

[12] I. Pill and T. Quaritsch. Behavioral diagnosis of LTL specifications at operator level. In *International Joint Conference on Artificial Intelligence*, pages 1053–1059, 2013.

[13] I. Pill and T. Quaritsch. Exploiting parse trees in LTL specification diagnosis. In *24th International Workshop on Principles of Diagnosis*, pages 59–64, 2013.

[14] Python Software Foundation. Python.org website, May 2014. https://www.python.org.

[15] SciPy Developers. Scipy website, May 2014. http://www.scipy.org.

[16] D. Abrahams. Boost.python, May 2014. http://www.boost.org/doc/libs/1_55_0/libs/python/doc/.

[17] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[18] DX-Competition Organizers. The 4th international diagnostic competition: DXC-2013, May 2014. http://www.dxc-2013.org.

[19] Gnuplot Authors. gnuplot homepage, May 2014. http://www.gnuplot.info.

[20] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[21] F. Wotawa. A variant of Reiter's hitting-set algorithm. *Information Processing Letters*, 79:45–51, 2001.

[22] B. Dutertre and L. de Moura. The YICES SMT solver. http://yices.csl.sri.com/tool-paper.pdf, 2006.

[23] N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(3-4):1–25, 2006.

[24] K. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[25] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. *Theory and Applications of Satisfiability Testing – SAT 2009*, 5584:167–180, 2009.

[26] L. Lin and Y. Jiang. The computation of hitting sets: review and new algorithms. *Information Processing Letters*, 86:177–184, 2003.

[27] R. Abreu and A. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Symposium on Abstraction Reformulation, and Approximation*, 2009.

[28] M. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design and Test*, 6:72–80, 1999. (http://www.cbl.ncsu.edu/benchmarks/ISCAS85).

[29] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[30] A. Metodi. Scryptominisat, May 2014. http://amit.metodi.me/research/scrypto/.

[31] M. Soos. Cryptominisat2, May 2014. http://www.msoos.org/cryptominisat2/.