Parallel Model-based Diagnosis on Multi-Core Computers

Dietmar Jannach
TU Dortmund, Germany
dietmar.jannach@tu-dortmund.de

Invited Talk at PCR workshop at CADE 2017

Introduction

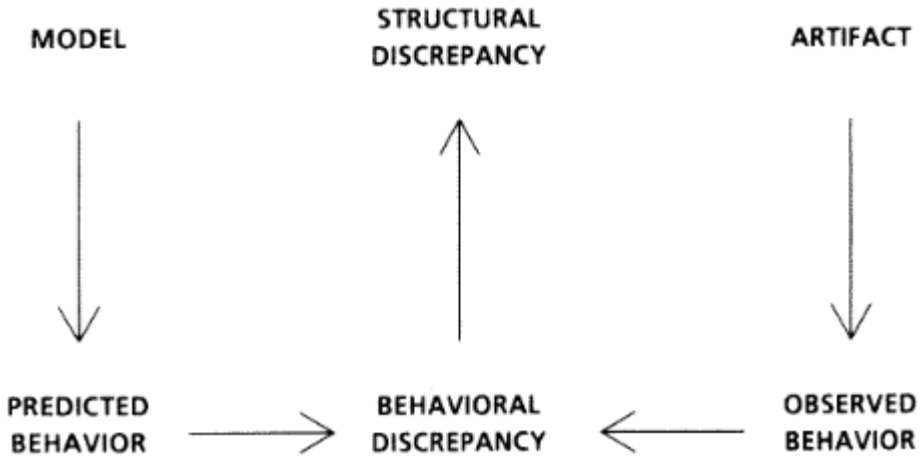
- Research interests
  - **Recommender Systems**
    - E-Commerce applications, business value of recommenders
    - Interactive advisory systems
  - Artificial Intelligence
    - Model-based Diagnosis, Constraints
  - **Software Engineering**
    - **Debugging of Spreadsheets**
    - **Drives this research (assuming a few cores)**
- **MBD for spreadsheets can be challenging**

Model-based Diagnosis (MBD)

- A subfield of Artificial Intelligence
- Concerned with the automated and principled analysis of why a system under observation does not work as expected
- Based on an explicit model of a system's behavior when all of its components work correctly
- Originally designed for diagnosis of hardware circuits
  - But applied in many other domains later on, in particular to software specifications:
    - Knowledge-base debugging, diagnosis of workflow definitions, VHDL and Java code, ontologies and description logics, spreadsheets

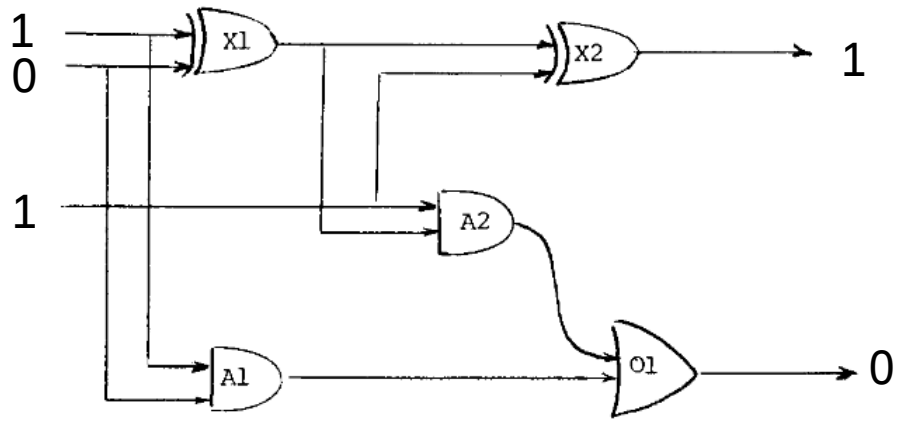
General Principle of MBD

- Detect and analyze the behavioral discrepancy
- Systematically test hypothesis about possible reasons for the discrepancy



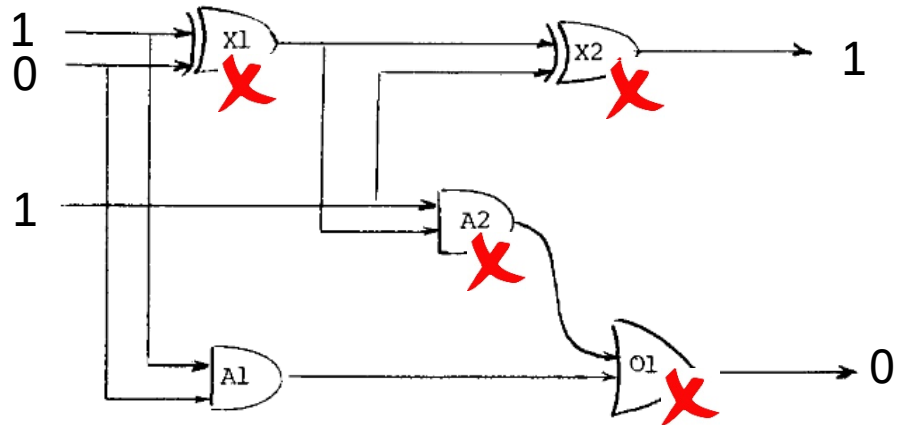
Diagnosing Electronic Circuits

- Given the inputs and observed outputs below, some components must be at fault
- The goal is to find possible (and parsimonious) explanations for the observed outputs



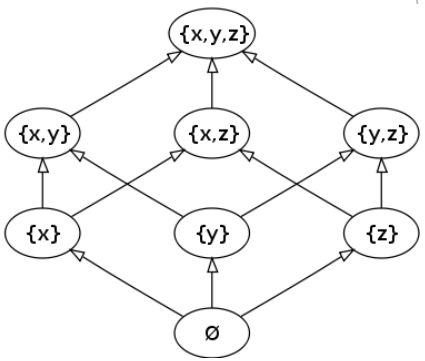
Diagnosing Electronic Circuits

- Assuming “everything is broken” is one possible explanation (diagnosis)
- But we are interested in minimal diagnosis
- A diagnosis is a subset of the system’s components which, if assumed faulty, explain (or: are logically consistent with) the observations



Diagnosis Algorithms

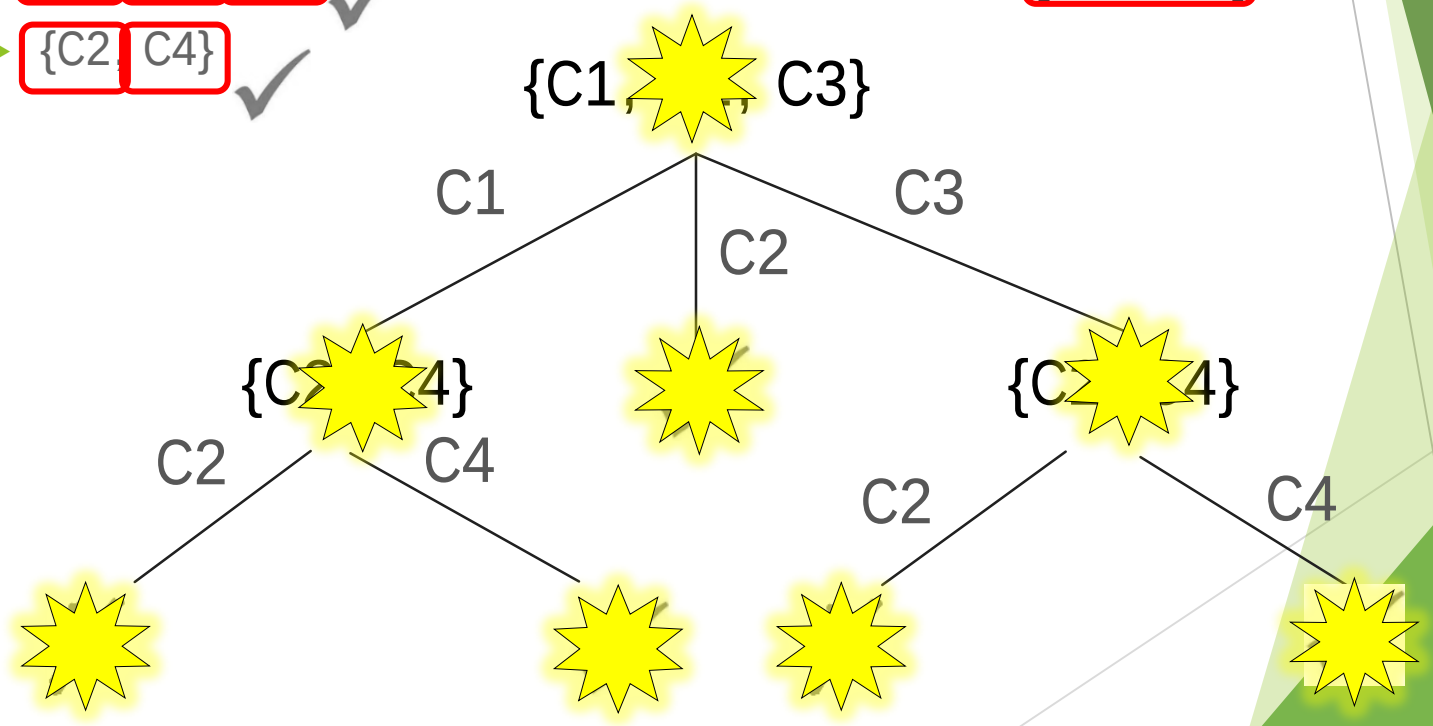
- A brute force algorithm
  - Test all hypothesis regarding the (two) possible states of each components
  - Means testing $\mathbf{2^n}$ combinations given **n** components to find all explanations
  - Each test involves a “simulation” of the system



- Reiter’s HS-Tree algorithm
  - Based on the concept of “conflicts”
    - Subsets of the components which cannot be assumed to work correctly
  - Conflicts guide the construction of a search tree
  - Prunes the search space significantly
  - Creates the diagnoses with increasing cardinality

Reiter's HS-Tree Algorithm

- Example
- Conflicts:
  - Not known in advance
  - {C1, C2, C3}
  - {C2, C4}
- Diagnoses:
  - {C2}
  - {C1, C4}
  - {C3, C4}

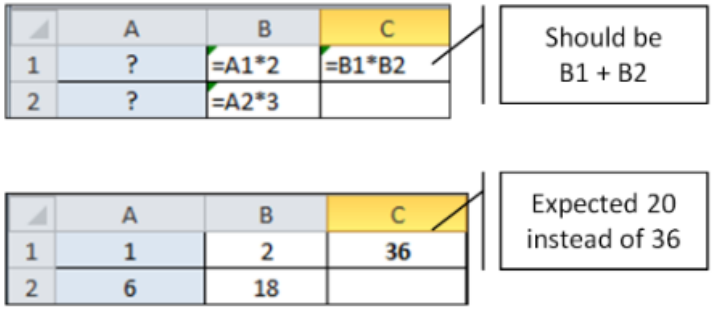


Reiter's Problem Formalization

- Sets SD, COMPONENTS, OBS
  - Can be encoded as sets of logical sentences
- Diagnosis problem: observation $o \in$ OBS deviates from expected system behavior
- Find diagnoses $\Delta \subseteq$ COMPONENTS that explain the systems behavior, if the components of Δ are assumed to be faulty
- Use HS-Tree algorithm to find minimal diagnoses
  - Based on conflicts
    - Conflict $c \subseteq$ COMPONENTS is a set of components that, if assumed to behave normally, are not consistent with the observations

Where's the constraint reasoning?

1. In many proposals constraint reasoning is used to simulate the system behavior
   - Own recent work – spreadsheet debugging



   - Spreadsheets are translated into a CSP program

2. Alternative approach: "Direct Diagnosis"
   - Don't use conflicts but encode the fault states into the simulation model

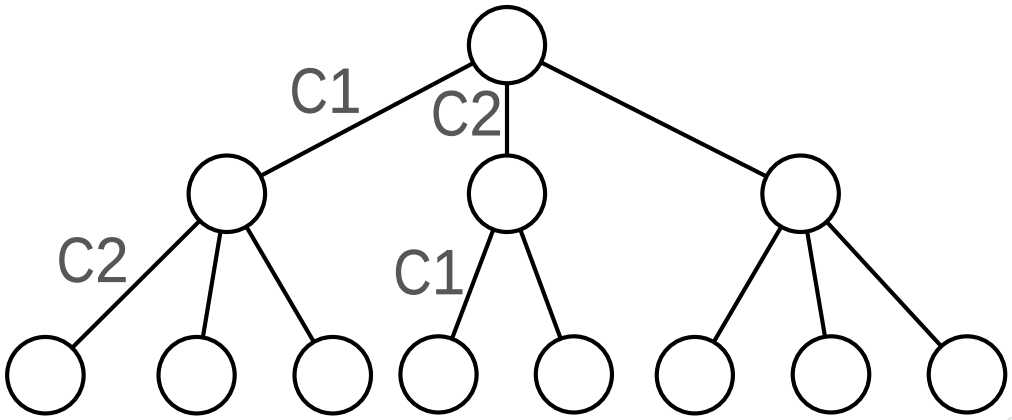
- Jannach, D. and Schmitz, T.: *"Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach"*. **Automated Software Engineering**, Vol. 23(1). Springer Nature, 2014, pp. 105-144

Computational Complexity

- Even if the conflicts were known in advance, the problem is hard
  - Reiter shows that the computation of the diagnoses corresponds to the computation of the “hitting sets” (cover set) of the conflict sets
  - Which is known to be an NP-hard problem
- Computing one additional node in the pruned search tree is costly as well
  - It can involve solving a given Constraint Satisfaction Problem multiple times
- **Our main proposal therefore**
  - Parallelize Reiter’s tree search algorithm (and thus implicitly the constraint reasoning process)
  - For some reason nobody thought of this
  - No parallel search in one CSP but many parallel CSPs

Level-wise parallelization

- Construct nodes at one level in parallel
  - Using thread pool of defined size
  - Synchronize at end of each level
- Limited synchronization effort needed
- Sound and complete



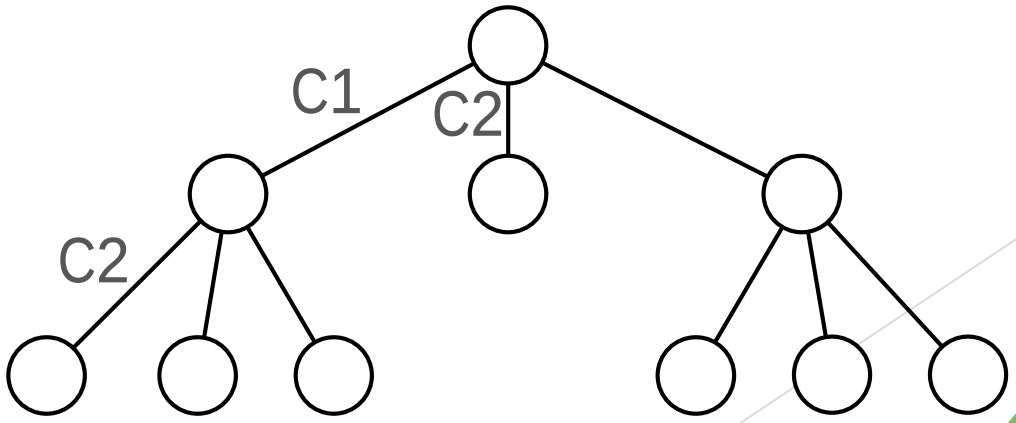
Jannach, D., Schmitz, T. and Shchekotykhin, K.: *"Parallel Model-Based Diagnosis on Multi-Core Computers"*. Journal of Artificial Intelligence Research, Vol. 55. AI Access Foundation, 2016, pp. 835-887

Full parallelization

- Except root node, all nodes are processed in parallel
- More synchronization required
  - Supersets of diagnoses can be found
  - When a diagnosis is found, supersets of this diagnosis have to be removed
- Sound and complete

Diagnoses:

~~{C1, C2}~~ {C2} …



Empirical Evaluation

- Tested different algorithm configurations on a number of datasets
- DXC Benchmarks
  - First 5 systems of 2011 DX Competition synthetic track, encoded as CSP problems
- CSPs
  - Selected CSPs of 2008 CSP Solver Competition, selected CSP-encoded spreadsheets
- Ontologies
  - Cannot be efficiently encoded as CSP problems
- Simulation
  - Evaluation based on a systematic variation of problem characteristics

DXC Benchmark

- DXC Synthetic Track
  - Real-world logic circuits
- First 5 systems
  - System specifies system description and components
- 20 scenarios per system
  - Scenario specifies observations
  - Different faulty components resulting in different observations
- 100 runs per scenario to factor out random effects
  - 5 x 20 x 100 x 3 = 30.000 runs

DXC Benchmarks

System	#C	#V	#F	#D	⌀#D	⌀|D|
74182	21	28	4 - 5	30 - 300	139	4.66
74L85	35	44	1 - 3	1 - 215	66.4	3.13
74283	38	45	2 - 4	180 - 4,991	1,232.7	4.42
74181*	67	79	3 - 6	10 - 3,828	877.8	4.53
c432*	162	196	2 - 5	1 - 6,944	1,069.3	3.38

System	Abs. seq. [ms]	LW-P		F-P	
		S_4	E_4	S_4	E_4
74182	78	1.95	0.49	1.78	0.44
74L85	209	2.00	0.50	2.08	0.52
74283	152,835	1.52	0.38	2.32	0.58
74181*	14,534	1.79	0.45	3.44	0.86
c432*	64,150	1.28	0.32	2.86	0.71

* limited the search depth to their actual number of faults to ensure termination within reasonable time frame

CSPs

- Different CSPs, different characteristics
  - Some can be solved in milliseconds
  - Others need hours or days
- Find CSPs that are
  - Different in their characteristics
  - Can be solved in a reasonable time frame
  - Not too simple
- Injected faults
  - To remove solvability with respect to test cases
  - Diagnosis task is to restore solvability

CSPs

Scenario	#C	#V	#F	#D	⌀ |D|	⌀ST
aim-50-1-6-3	130	100	5	12	3	< 1
c8	523	239	8	4	6.25	< 1
costasArray-13	87	88	2	2	2.5	~ 3
domino-100-100	100	100	3	81	2	< 1
e0ddr1-10-by-5-8	265	50	17	15	4	~ 35
fischer-1-1-fair	320	343	9	2006	2.98	~ 10
graceful–K3-P2	60	15	4	117	2.94	~ 1.5
graph2	2245	400	14	72	3	~ 140
mknap-1-5	7	39	1	2	1	< 1
primes-15-20-3-1	20	100	3	2	1	< 1
queens-8	28	8	15	9	10.9	< 1
series-13	156	25	2	3	1.3	~ 150

CSPs

Scenario	Abs. seq. [ms]	LW-P		F-P	
		S_4	E_4	S_4	E_4
aim-50-1-6-3	5,245	2.09	0.52	2.32	0.58
c8	1,685	1.08	0.27	1.22	0.30
costasArray-13	7,367	1.61	0.40	1.79	0.45
domino-100-100	8,628	1.68	0.42	1.77	0.44
e0ddr1-10-by-5-8	5,875	2.34	0.59	2.31	0.58
fischer-1-1-fair	422,559	1.17	0.29	1.18	0.29
graceful–K3-P2	3,480	2.29	0.57	2.30	0.58
graph-2	94,398	1.87	0.47	1.72	0.43
mknap-1-5	383	1.26	0.31	1.30	0.33
primes-15-20-3-1	323	1.31	0.33	1.29	0.32
queens-8	4,824	1.72	0.43	2.10	0.52
series-13	7,432	1.82	0.46	1.48	0.37

Simulation

- Test effects of different characteristics on parallelization improvements
- Simulation
  - Artificial problem settings
  - Defined problem characteristics
  - Randomly created problem instances
  - Whenever a new conflict should be determined, system actively waits some time (Wt) and randomly returns one of the conflicts

Simulation Results

#Cp, #Cf, $\overline{\lvert Cf \rvert}$	#D	Wt [ms]	Seq. [ms]	LWP		FP	
				S_4	E_4	S_4	E_4
Varying computation times Wt							
50, 5, 4	25	**0**	23	2.26	0.56	**2.58**	**0.64**
50, 5, 4	25	**10**	483	2.98	0.75	**3.10**	**0.77**
50, 5, 4	25	**100**	3,223	2.83	0.71	**2.83**	**0.71**

- Quite small diagnosis problem
- Wt = 0 shows time for tree construction itself
  - Synchronization overhead of Full Parallelization

More Simulation Results

- Other results
  - Larger conflicts → broader HS-Trees → better parallelization
  - More components → higher problem complexity → narrower HS-Trees up to a certain level → smaller parallelization improvements
  - Adding more threads → even higher improvements, but efficiency decreases

Computing multiple conflicts at once

- When constructing a new node, exactly one minimal conflict is computed
- Pro:
  - The new conflict is quickly visible and can be used by parallel threads
- Con:
  - Conflict search is re-started for each node
- Approach:
  - New method (MergeXPlain) to compute more than one conflict, in case they exist
- Effect:
  - Slightly more effort for first nodes, but higher re-use levels later on

Shchekotykhin, K., Jannach, D. and Schmitz, T.: *"MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis"*. In: IJCAI 2015. Buenos Aires, Argentina, 2015, pp. 3221-3228

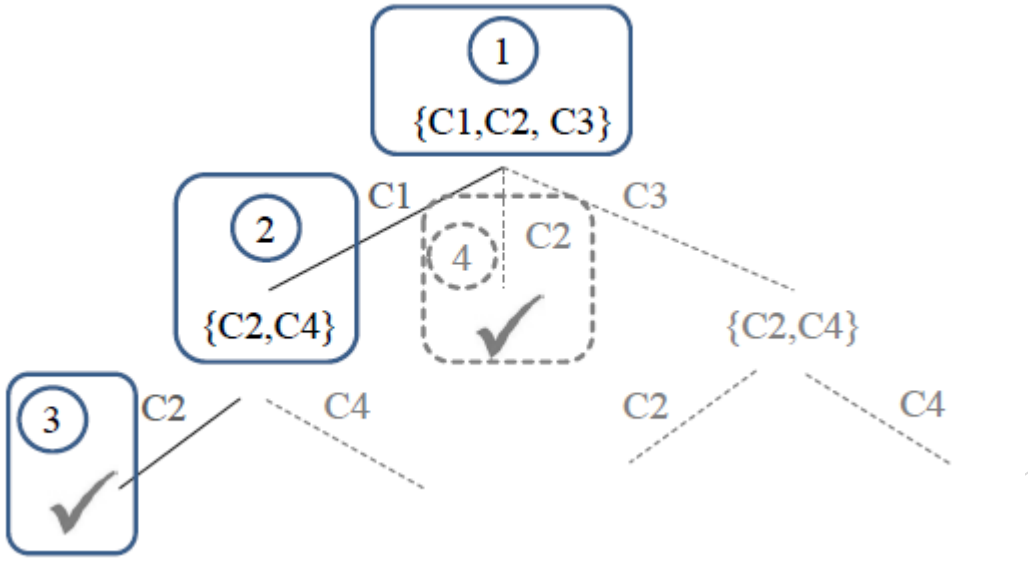
Evaluation

- Different technical implementations possible
  - Compute all conflicts and then return
  - Return to main thread after first conflict is found
    - And compute more in a new background thread
- Results (MergeXPlain combined with parallelization)

System	Seq.(QXP) [ms]	FP(QXP) S_4	FP(QXP) E_4	Seq.(MXP) [ms]	FP(MXP) S_4	FP(MXP) E_4
74182	12	1.26	0.32	10	**1.52**	**0.38**
74L85	15	1.36	0.34	12	**1.33**	**0.33**
74283	49	1.58	0.39	35	**1.48**	**0.37**
74181	699	1.99	0.55	394	**2.10**	**0.53**
c432	3,714	1.77	0.44	2,888	**1.72**	**0.43**

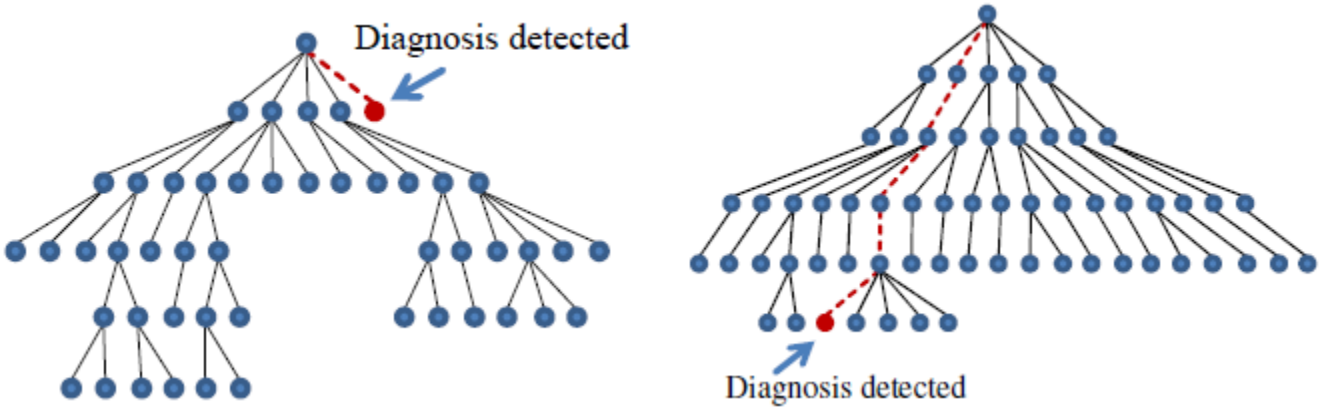
Parallelizing Depth-First Search

- Parallelizing a DFS procedure to find **one** diagnosis
- Go down the tree in a random manner in parallel threads
  - Remove redundant elements once a diagnosis is found



A Hybrid DFS/BFS Strategy

- Which strategy works best depends on the specific problem setting



Results DFS/BFS Strategy

- Electronic circuits as an example
- Randomized DFS-strategy works better

System	Seq.	FP		RDFS	PRDFS		Hybrid	
	[ms]	S_4	E_4	[ms]	S_4	E_4	S_4	E_4
74182	16	1.37	0.34	**9**	0.84	0.21	0.84	0.21
74L85	13	**1.34**	**0.33**	11	1.06	0.27	1.05	0.26
74283	54	1.67	0.42	25	**1.22**	**0.31**	1.06	0.26
74181	691	2.08	0.52	74	**1.23**	**0.31**	1.04	0.26
c432	2,789	1.89	0.47	1,435	**2.96**	**0.74**	1.81	0.45

Direct Encodings

- Assume a CSP with variables: a1, a2, b1, b2, c1
- Constraints are as follows
  - X1 : b1 = a1 * 2; X2 : b2 = a2 * 3; X3 : c1 = b1 x b2
  - But X3 should have been: c1 = b1 + b2
- In a direct encoding, we add health state variables for each constraint (the constraints are the components), i.e., ab_1, ab_2, ab_3
- Updated constraints are
  - X1: ab_1 v (b1 = a1 * 2); X2: ab_2 v (b2 a2 * 3); X3: ab_3 v (c1 = b1 * b2)
- Add: $ab_1 + ab_2 + ab_3 = 1$

Direct Encodings

- Very fast at finding one diagnosis
- All diagnosis can be obtained by incrementing the expected diagnosis size stepwise
- Using parallel constraint reasoning implementation of Gecode solver

Direct Encoding Results

- Finding one or all diagnosis
- Parallelization in both cases starts paying off for more complex problems
- Might be even better if specifics of the solver are taken into account

System	Direct Encoding				
	Abs. [ms]	S_2	E_2	S_4	E_4
74182	**27**	0.85	0.42	0.79	0.20
74L85	**30**	0.89	0.44	0.79	0.20
74283	**32**	0.85	0.43	0.79	0.20
74181	200	1.04	0.52	**1.15**	0.29
c432	1,399	1.17	0.58	**1.25**	0.31

Summary

- Model-based Diagnosis as a general fault detection/isolation method
- Based on a simulation of the system to be examined
- Simulation (or problem itself) often a CSP
- Our work shows that parallelizing the diagnostic process leads to significant performance improvements
  - Multiple CSP problems solved in parallel
- Parallelizing direct encodings also leads to performance gains

Thank you for your attention

Contact: dietmar.jannach@tu-dortmund.de

References

- Jannach, D., Schmitz, T. and Shchekotykhin, K.: *"Parallel Model-Based Diagnosis on Multi-Core Computers"*. **Journal of Artificial Intelligence Research**, Vol. 55. AI Access Foundation, 2016, pp. 835-887
- Jannach, D. and Schmitz, T.: *"Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach"*. **Automated Software Engineering**, Vol. 23(1). Springer Nature, 2014, pp. 105-144
- Shchekotykhin, K., Jannach, D. and Schmitz, T.: *"MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis"*. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (**IJCAI 2015**). Buenos Aires, Argentina, 2015, pp. 3221-3228
- Jannach, D., Schmitz, T. and Shchekotykhin, K. M.: *"Parallelized Hitting Set Computation for Model-Based Diagnosis"*. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015). Austin, Texas, USA, 2015, pp. 1503-1510

HS-Tree algorithm

Algorithm 1: DIAGNOSE: Main algorithm loop.

Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set Δ of diagnoses

```
1 Δ = ∅; paths = ∅; conflicts = ∅;
2 nodesToExpand = ⟨GENERATEROOTNODE(SD, COMPS, OBS)⟩;
3 while nodesToExpand ≠ ⟨ ⟩ do
4     newNodes = ⟨ ⟩;
5     node = head(nodesToExpand) ;
6     foreach c ∈ node.conflict do
7         GENERATENODE(node, c, Δ, paths, conflicts, newNodes);
8     nodesToExpand = tail(nodesToExpand) ⊕ newNodes;
9 return Δ;
```

Expansion logic

Algorithm 2: GENERATENODE: Node generation logic.

Input: An *existingNode* to expand, a conflict element $c \in$ COMPS, the sets Δ, *paths*, *conflicts*, *newNodes*

```
1  newPathLabel = existingNode.pathLabel ∪ {c};
2  if (∄ l ∈ Δ : l ⊆ newPathLabel) ∧ CHECKANDADDPATH(paths, newPathLabel) then
3      node = new Node(newPathLabel);
4      if ∃ S ∈ conflicts : S ∩ newPathLabel = ∅ then
5          node.conflict = S;
6      else
7          newConflicts = CHECKCONSISTENCY(SD, COMPS, OBS, node.pathLabel);
8          node.conflict = head(newConflicts);
9      if node.conflict ≠ ∅ then
10         newNodes = newNodes ⊕ ⟨node⟩;
11         conflicts = conflicts ∪ newConflicts;
12     else
13         Δ = Δ ∪ {node.pathLabel};
```

Level-wise Parallelization

Algorithm 4: DIAGNOSELW: Level-Wise Parallelization.

Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set Δ of diagnoses

```
1  Δ = ∅; conflicts = ∅; paths = ∅;
2  nodesToExpand = ⟨GENERATEROOTNODE(SD, COMPS, OBS)⟩;
3  while nodesToExpand ≠ ⟨ ⟩ do
4      newNodes = ⟨ ⟩;
5      foreach node ∈ nodesToExpand do
6          foreach c ∈ node.conflict do          // Do computations in parallel
7              threads.execute(GENERATENODE(node, c, Δ, paths, conflicts, newNodes));
8      threads.await();                          // Wait for current level to complete
9      nodesToExpand = newNodes;                 // Prepare next level
10 return Δ;
```

Full parallelization

Algorithm 5: DIAGNOSEFP: Full Parallelization.

Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set Δ of diagnoses

```
1  Δ = ∅; paths = ∅; conflicts = ∅;
2  nodesToExpand = ⟨GENERATEROOTNODE(SD, COMPS, OBS)⟩;
3  size = 1; lastSize = 0;
4  while (size≠lastSize) ∨ (threads.activeThreads≠ 0) do
5      for i = 1 to size − lastSize do
6          node = nodesToExpand.get[lastSize + i];
7          foreach c ∈ node.conflict do
8              threads.execute(GENERATENODEFP(node, c, Δ, paths, conflicts,
                 nodesToExpand));
9      lastSize = size;
10     wait();
11     size = nodesToExpand.length();
12 return Δ;
```

Full-parallelization node expansion

Algorithm 6: GENERATENODEFP: Extended node generation logic.

Input: An *existingNode* to expand, $c \in$ COMPS,
sets Δ, *paths*, *conflicts*, *nodesToExpand*

```
1  newPathLabel = existingNode.pathLabel ∪ {c};
2  if (∄ l ∈ Δ : l ⊆ newPathLabel) ∧ CHECKANDADDPATH(paths, newPathLabel) then
3      node = new Node(newPathLabel);
4      if ∃ S ∈ conflicts : S ∩ newPathLabel = ∅ then
5          node.conflict = S;
6      else
7          newConflicts = CHECKCONSISTENCY(SD, COMPS, OBS, node.pathLabel);
8          node.conflict = head(newConflicts);
9      synchronized
10         if node.conflict ≠ ∅ then
11             nodesToExpand = nodesToExpand ⊕ ⟨node⟩;
12             conflicts = conflicts ∪ newConflicts;
13         else if ∄ d ∈ Δ : d ⊆ newPathLabel then
14             Δ = Δ ∪ {node.pathLabel};
15             for d ∈ Δ : d ⊇ newPathLabel do
16                 Δ = Δ \ d;
17 notify();
```