Heuristics for Hierarchical Partitioning with Application to Model Checking

M. Oliver Möller
Rajeev Alur

BRICS Report Series RS-00-21

ISSN 0909-0878 August 2000

Heuristics for Hierarchical Partitioning with Application to Model Checking

M. Oliver Möller[†] and Rajeev Alur[‡]

[†] BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Århus C, Denmark
`omoeller@brics.dk`

[‡] **University of Pennsylvania**
Computer & Information Science
Moore building
200 South 33rd Street
Philadelphia, PA 19104, USA
`alur@cis.upenn.edu`

30 August 2000

Abstract

Given a collection of connected components, it is often desired to cluster together parts of strong correspondence, yielding a hierarchical structure. We address the automation of this process and apply heuristics to battle the combinatorial and computational complexity.

We define a cost function that captures the quality of a structure relative to the connections and favors shallow structures with a low degree of branching. Finding a structure with minimal cost is shown to be *NP*-complete. We present a greedy polynomial-time algorithm that creates an approximative good solution incrementally by local evaluation of a heuristic function. We compare some simple heuristic functions and argue for one based on four criteria: The number of enclosed connections, the number of components, the number of touched connections and the depth of the structure.

We report on an application in the context of formal verification, where our algorithm serves as a preprocessor for a temporal scaling technique, called *"Next" heuristic* [AW99]. The latter is applicable in enumerative reachability analysis and is included in the recent version of the Mocha model checking tool.

We demonstrate performance and benefits of our method and use an asynchronous parity computer, a standard leader election algorithm, and an opinion poll protocol as case studies.

1 Introduction

Detailed descriptions of large systems are often too complex to be comprehensive. It is not sufficient to have complete explicit data. In order to make sense out of vast amounts of information, we need structure. As humans, we have limited ability to mentally deal with a large number of objects simultaneously. Structure helps us to focus on essential parts and devise a reasonable order in which to proceed. Perhaps surprisingly, this also holds true for machines, whenever the data has to be processed in a nontrivial way.

Consider a system with 5 components A, B, C, D, E. Suppose components A and B are connected (for instance, could share a variable in the context of communicating processes), components A, B, C are connected, B and D are connected, and so are D and E. Then, instead of viewing the system as a *set* of 5 components, we may want to view it as a tree $\{\{\{A,B\},C\},\{D,E\}\}$. In this structuring, only the connection between B and D needs to be visible (or understood) at the root. We will formalize the input as a hypergraph, structuring as a hierarchical partitioning, and study the problem of designing a *good* structuring.

A *hypergraph* $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ is a finite set of vertices $\mathcal{C}$ together with a multi set $\mathcal{E}$, where every *hyperedge* $e \in \mathcal{E}$ is a subset of $\mathcal{C}$. We assume that every e corresponds to a unique label ℓ_e. Hyperedges of size 0 or 1 are disallowed. When drawing hypergraphs, we follow the edge standard, that coincides with the common graph representation for the special case, that every hyperedge is of size 2.[1]

We describe *hierarchical partitionings* as trees over leaf nodes $\mathcal{C}$, where all internal nodes are assumed to have degree at least 2. As a convention, we draw these trees over a given hypergraph by introducing one polyhedron for each internal node. All enclosed polyhedrons and vertices in $\mathcal{C}$ are children of this node.

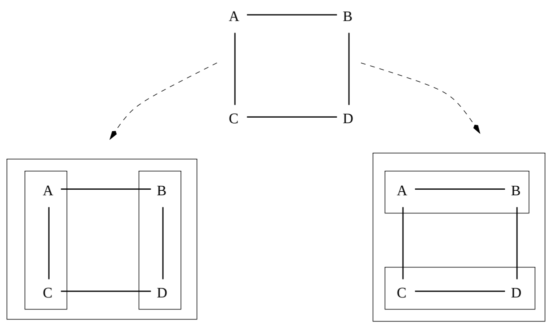


Figure 1: Two symmetric ways to structure a square.

Qualifying hierarchical structures Often there are various hierarchical partitionings we would consider qualitatively equivalent. If symmetries—in terms of vertex permutations that map one structure to the other—can be found, we want to rate them equally high. In Figure 1, the four atomic components A, B, C, D are connected via the hyperedges $\{A,B\}$, $\{A,C\}$, $\{B,D\}$, and $\{C,D\}$. The left structure can be mapped into the right one

[1]For a overview on graphs and hypergraphs see e.g. [Ber73].

by rotating the atoms counter-clockwise, thus these structures are symmetric. This leaves open the question as how to find symmetries efficiently.[2]

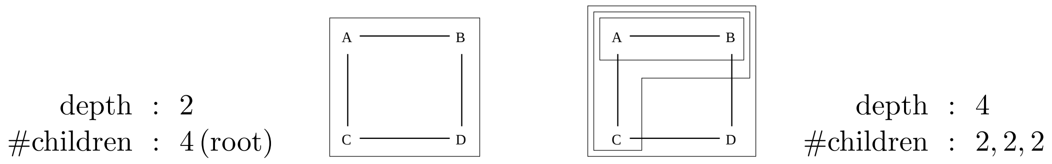


Figure 2: Two less desirable hierarchical partitionings of the same square.

The way to qualify is heavily dependent on the application we have in mind, but there are two natural properties we tend to prefer: Low depth and low degree of branching. In the example in Figure 1, the hierarchy of each partitioning is of depth 2 and the degree of branching on each level is 2 as well. Figure 2 depicts two less desirable partitionings. The left tree is of depth 1, but the root has 4 children. On the right we see a hierarchical partitioning of depth 3 with branching degree 2. In general, the two criteria are competing goals.

Plan This report is organized as follows. The next section explains a motivation for hierarchical partitioning in model checking. In section 3 we define and characterize the general problem. In section 4 we develop a greedy algorithm to generate approximative good solutions. Section 5 reports on experiments with sample problems. Section 6 reflects on the limitation of our method, contrasts it to related work and lists open problems. We end with a summary.

2 Hierarchical Partitioning for Formal Verification

The problem of finding apt hierarchical partitionings has concrete applications in formal verification. Here we are often confronted with collections of entities or processes, that are interconnected via channels or shared variables. Typically, the communication structure offers asymmetries that can be exploited by means of introducing property-preserving abstractions of sub-components.

Our specific motivation is a safety-property preserving optimization of a model checking algorithm. *Model checking* [CE82, CK96, Hol97] is widely accepted as a useful technique for the formal verification of high-level designs in hardware and communication protocols. Since it typically requires search in the global state-space, much research aims at providing heuristics to make this step less time- and space-consuming.

Consider 4 processes A, B, C, D such that A and B communicate using channel x, A and C communicate using channel y, C and D communicate using channel z and D and B

[2]In order to identify symmetries, we have to solve a special graph isomorphism problem. Though this is know to be in *NP* and unlikely to be *NP*-complete, no polynomial-time algorithm is known, see [KST93].

communicate using channel u. This corresponds to the hypergraph of Figure 1. Instead of viewing the system as a set of 4 processes, if we view the system hierarchically decomposed as shown in left of Figure 1, tools such as concurrency workbench [CPS91, CPS93], can analyze it in the following way. First take the product of processes A and C. Now the communication channel y can be viewed as internal to this composite process, and we can apply a reduction based on weak bisimulation minimizing the size. Analogously, compose B and D, and minimize considering the channel u.

The hierarchical partitioning is exploited in a different way in the model checker MOCHA, and will form the basis of the experiments in this paper. In the context of reactive modules [AH96], a specialized heuristic was proposed and implemented in the recent version of this verification tool: The "Next" heuristic [AW99], that allows to ignore irrelevant transitions by temporal scaling. Sequences of transitions are compressed into a single meta-transition, thus saving time and memory in the state space exploration. The technique assumes an initial structure of the original system as isolated processes called modules. It is applicable whenever the modules are enabled to move asynchronously and the formula in question is a safety property. By clustering modules together, we can hide local behavior via hiding variables that cannot affect the remaining system. Thus we are allowed to replace a sub-system P of by an abstracted version, called *next* Θ *for* P, where Θ expresses changes that cannot be hidden. This technique can be applied in a hierarchical manner.

The gain depends strongly on the structure we impose, i.e. on the choices of P. As a rule, we want to hide variables as soon as possible. At the same time it is an advantage, if the structure reflects areas of strong interaction, i.e. if many variables can be hidden in sub-components at the same time. It is to be expected, that even in good partitioning some variables cannot be hidden early. The challenge is to make advantageous choices. In last consequence, the quality of our structure is determined by the savings with respect to a model-checking algorithm. However, this not an applicable measure, since predicting the run-time is in general not easier that solving the model-checking problem itself.[3] Therefore we introduce an approximate measure, that relies only on syntactic properties of the given system.

In the following we consider a system to be given as a hypergraph, where processes are vertices and shared variables among them are represented by hyperedges. We rearrange the vertices as leaves of a rooted tree $\mathcal{T}$. Hyperedges that range over large portions of the tree are punished in terms of cost. The problem of structuring the system reduces then to find a tree of low cost.

[3]Assume the desired property does not to hold. Then the best structure uncovers a short violating path, which is much cheaper than an exhaustive state space exploration. We cannot know that this path exists without solving the original problem, thus we cannot have a tight estimate on the run-time without also knowing the answer.

3 The Tree-Indexing Problem

We want to restructure a given hypergraph in a hierarchical manner. A *tree-indexing* $\mathcal{T}$ of a hypergraph $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ is a rooted tree over leaf nodes $\mathcal{C}$, where every internal node has at least two children. Every tree-indexing corresponds to a cost value dependent on $\mathcal{E}$.

n	1	2	3	4	5	6	7	8	9	10
#tree-indexings $T(n)$	1	1	4	26	236	2'752	39'208	660'032	12'818'912	282'137'824

Table 1: The combinatorial explosion in the number of tree-indexings.

Combinatorial Complexity Given a hypergraph with n labeled vertices, we want to determine the number $T(n)$ of distinguishable tree-indexings. This is in an equivalent formulation recorded as Schröder's fourth problem [Sch70]. It can be solved (for every fixed n) via a generating function method. Let $\varphi(z)$ be the ordinary generating function, where the n^{TH} coefficient corresponds to $T(n)$. Let $\hat{\varphi}(z)$ be its exponential transform. Since every structure is either atomic or a set of smaller structures (with cardinality at least two), $\hat{\varphi}(z)$ can be expressed according to the theory of admissible constructions [FZV94, Fla88] as follows:

$$\hat{\varphi}(z) \;=\; \mathit{Union}\,(z, \mathit{Set}\,(\hat{\varphi}(z), \text{cardinality} \geq 2))$$

This transcribes to

$$\begin{aligned} && \hat{\varphi}(z) &= z + \sum_{k\geq 2} \tfrac{1}{k!} \cdot (\hat{\varphi}(z))^k \\ \Leftrightarrow && \hat{\varphi}(z) &= z + \exp(\hat{\varphi}(z)) - \hat{\varphi}(z) - 1 \\ \Leftrightarrow && \exp(\hat{\varphi}(z)) &= 2\,\hat{\varphi}(z) - z + 1 \end{aligned}$$

There is no closed form known for $\hat{\varphi}(z)$, $\varphi(z)$, or $T(n)$. However, for every fixed n we can extract the n^{TH} coefficient of $\varphi(z)$ with algebraic methods and thus approximate $T(n)$ (see [Fla97]). Table 1 gives an impression how fast this series grows. Thus we have only little hope to perform an exhaustive search on the domain of possible tree-indexings.

Computational Complexity We can formulate the problem of finding a good tree-indexing as an optimization problem relative to a fixed *cost* function. This function should punish both deep structures and hyperedges that span over big subtrees. For every $e \in \mathcal{E}$ let $\mathcal{T}_e$ denote the smallest complete subtree of $\mathcal{T}$, such that every vertex $v \in e$ is a leaf of $\mathcal{T}_e$. With *leaves*$(\mathcal{T})$ we denote the set of leaf nodes in $\mathcal{T}$. The *depth* of $\mathcal{T}$ is the length of the longest descending path from its root. The *depth cost* of a tree $\mathcal{T}$ is defined as a function

$$\mathit{depth\_cost}(\mathcal{T}) \;:=\; \begin{cases} 2 & \text{if } \mathit{depth}(\mathcal{T}) = 1 \\ \mathit{depth}(\mathcal{T}) & \text{otherwise} \end{cases} \qquad (1)$$

The cost of a tree-indexing $\mathcal{T}$ is then defined relative to $\mathcal{H} = (\mathcal{C},\ \mathcal{E})$.

$$cost(\mathcal{T}) \quad := \quad \sum_{e \in \mathcal{E}} depth\_cost(\mathcal{T}_e) \ \cdot \ |leaves\,(\mathcal{T}_e)\,| \tag{2}$$

For example, both tree-indexings in Figure 1 have cost $2 \cdot 2 \cdot 2 + 2 \cdot 2 \cdot 4 = 24$, whereas the tree-indexings in Figure 2 are of cost $4 \cdot 2 \cdot 4 = 32$ and $2 \cdot 2 + 2 \cdot 3 + 2 \cdot 3 \cdot 4 = 34$ respectively.

EDGE-GUIDED TREE-INDEXING: Given a hypergraph $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ and a number $K \in I\!N$. Decide whether there exists a tree-indexing of cost at most K.

The problem EDGE-GUIDED TREE-INDEXING is *NP*-complete, even if we restrict to the special case where $\mathcal{H}$ is a multi graph. A proof by reduction from MINIMUM CUT INTO EQUAL-SIZED SUBSETS is given in Appendix A. We expect EDGE-GUIDED TREE-INDEXING to remain *NP*-hard for other non-trivial definitions of a cost function like $\sum_e |leaves(\mathcal{T}_e)|$, though we do not give a proof for this. This precludes the possibility to determine an *optimal* tree-indexing in polynomial time[4] and suggests the application of heuristics in order to find a reasonably good tree-indexing efficiently.

4 Heuristic Solutions

The key observation is that not all tree-indexings make sense. We want to group only components together, that are immediately (and possibly strongly) related. In Figure 1 no one would think of grouping A with D, since they do not immediately share any hyperedge. We can make use of this asymmetry in terms of simple strategies, that start with a flat structure and incrementally construct a hierarchical one. For simplicity we refer to intermediate structures always as forests over a fixed set of n leaf nodes.

4.1 Partitioning Pairwise

We propose two simple strategies that incrementally construct a tree-indexing by introducing one new root node in each step. Though they fail to yield desired results in simple scenarios, they guide us to the construction of an improved method.

Let us start with a forest, where every tree consists of a single node. Now we repeatedly combine two trees that share a hyperedge under a new root.

Strategy I: *Group together any pair, that shares a hyperedge.*

With this method, we avoid to combine unrelated sub-trees. Since with every step the number of trees in the forest decreases by one, this terminates after $n-1$ steps. Initially there are $\mathcal{O}(n^2)$ pairs to check and in any of the $n-1$ steps we have to consider at most n

[4]Unless *NP* turns out to be equal to *P*.

new candidates. Assuming we can check whether A and B share one hyperedge in constant time, Strategy I has run-time complexity $\mathcal{O}(n^2)$. It is easy to spot the shortcomings. It is barely checked, whether there is *any* connection between a considered pair, everything else is ignored. In Figure 3 we show a simple scenario, where Strategy I could yield the desired result on the left only by chance. The dashed outlines of polyhedra indicate, that no hyperedge can be hidden within these subtrees.

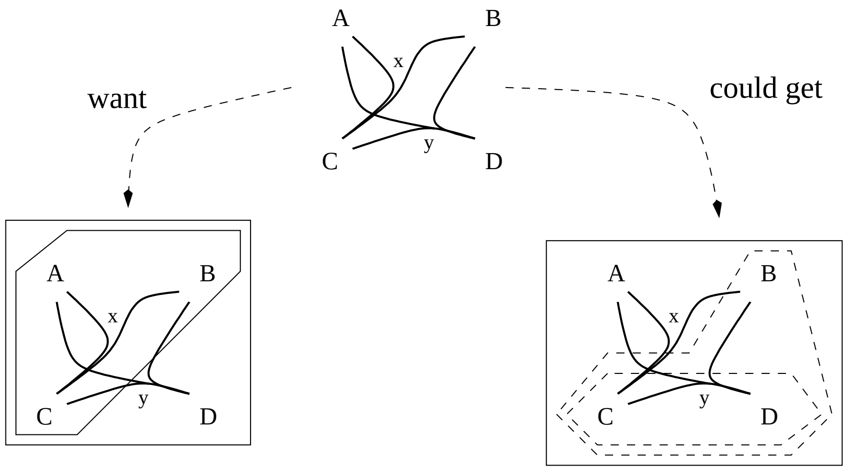


Figure 3: Strategy I can easily yield undesired results.

Though A and B were somehow stronger connected than A and D (by sharing two hyperedges instead of just one), our strategy was blind to that fact. So let us take this degree of connectedness into account.

Strategy II: *Group together the pair that shares the most hyperedges.*

This does not significantly increase the complexity, since we can maintain the number of pairs in an apt data structure, e.g., a priority queue sorted by the number of shared hyperedges. As we see in Figure 4, Strategy II yields more satisfactory structures.

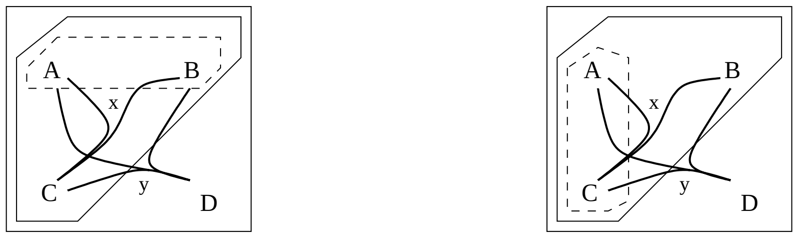


Figure 4: The two possible results when applying Strategy II.

However, it is not too difficult to find scenarios, where also Strategy II performs poorly, one is shown in Figure 5. Since the components C, D, E, F, G are connected via three hyperedges, the more intuitive subtree $\{A, B, C\}$ is never considered.

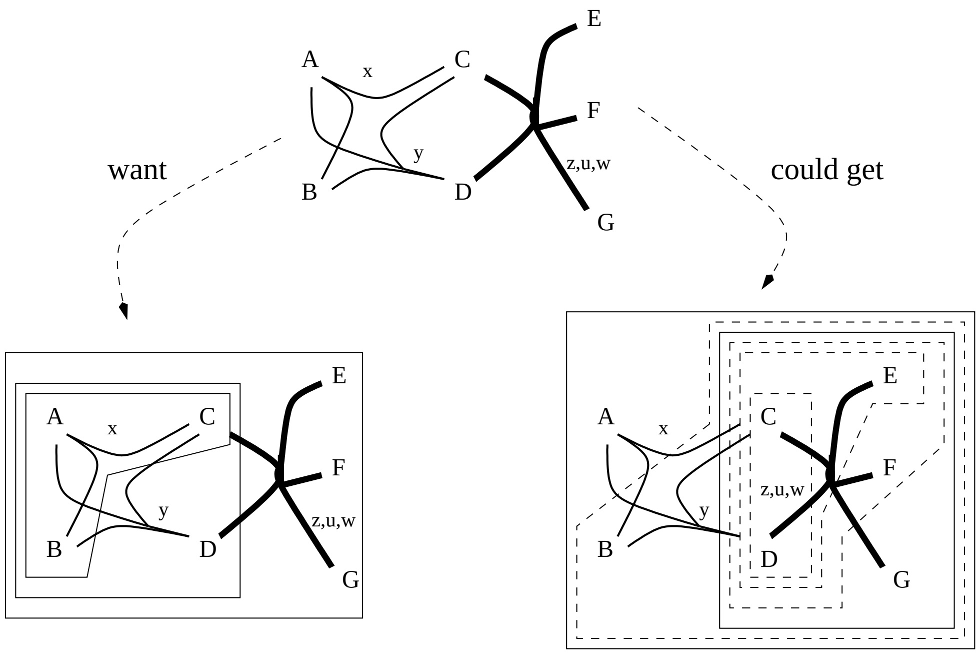


Figure 5: A way to fool Strategy II.

It is likely that we can come up with a bad scenario for *any* simple strategy, especially if the number of choices it considers is quite small. This serves as an argument to increase the size of our search space and consider also larger tuples as candidates.

4.2 A Greedy Algorithm for Partitioning

The simple ideas from the previous section can be extended to shape a greedy algorithm. Starting with hypergraph $\mathcal{H} = (\mathcal{C}, \mathcal{E})$, maintain a partial tree-indexing, i.e., a forest $\mathcal{F}$ with leaves $\mathcal{C}$. Repeatedly pick the best set of trees in $\mathcal{F}$ from a (possibly restricted) set of candidates. The heuristic component is the selection and comparison of candidates. The latter is realized via the pre-order induced by a rating function $\mathbf{r} : 2^{\mathcal{C}}_{\star} \times \wp^{\mathcal{D}} \to I\!R$, where $2^{\mathcal{C}}_{\star}$ denotes a multi set of hyperedges and $\wp^{\mathcal{D}}$ is the set of forests over leaves $\mathcal{D} \subseteq \mathcal{C}$. The higher the rating, the better the candidate.
The algorithm in Figure 6 proceeds as follows. $\mathcal{F}$ is initialized as the forest with $|\mathcal{C}|$ trees, each consisting of a single node. After inserting a set of candidates proposed for grouping together in a priority queue, a small number of executions of the while-loop follows. In each execution, the most promising candidate $\mathcal{A}$ is dequeued and the data is updated: The trees in $\mathcal{A}$ are replaced by a tree with the fresh root $\mathcal{A}'$ and children $t \in \mathcal{A}$. Every set

```
Algorithm: partition_incrementally
  input:   hypergraph $\mathcal{H} = (\mathcal{C},\ \mathcal{E})$
  output:  tree-indexing over leaves $\mathcal{C}$

  PriorityQueue $Q :=$ emptyQueue
  Forest $\mathcal{F} := \mathcal{C}$

  FORALL considered candidates $\mathcal{A} \subseteq \mathcal{F}$
      insert($\mathcal{A}$, $Q$)
  WHILE notempty($Q$)
      $\mathcal{A} :=$ top($Q$)
      let $\mathcal{A}' :=$ fresh node with children $t \in \mathcal{A}$
      $\mathcal{F} := (\mathcal{F} \setminus \mathcal{A}) \cup \{\mathcal{A}'\}$
      $\mathcal{E} := \mathcal{E} \setminus \{e \,|\, e \subseteq \mathcal{A}\}$   /* remove covered hyperedges */
      update($\mathcal{E}$, $\mathcal{A}$, $\mathcal{A}'$)      /* replace all $t \in \mathcal{A}$ by $\mathcal{A}'$ */
      FORALL $\mathcal{B} \in Q$ with $\mathcal{B} \cap \mathcal{A} \neq \emptyset$
          remove($\mathcal{B}$, $Q$)
      FORALL new candidates $\mathcal{D}$ containing $\mathcal{A}'$
          insert($\mathcal{D}$, $Q$)
  RETURN $\mathcal{F}$
```

Figure 6: Incremental algorithm for constructing a tree-indexing.

containing trees $t \in \mathcal{A}$ is removed from the priority queue and new candidates containing $\mathcal{A}'$ are inserted. Hyperedges $e \subseteq \mathcal{A}$ are deleted so that they do not influence later selections.

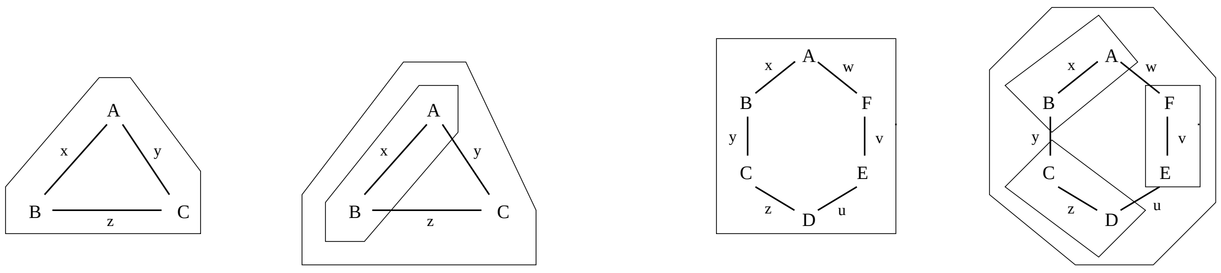
This description leaves open the questions, what should be used as a rating function and which candidates should be considered in first place. We explain these aspects of the algorithm in the following.

Developing a good rating function In a reasonable definition of a rating function, several factors concerning the structure of the proposed candidate have to be taken into account. Most importantly, we want to know the number of additional hyperedges that are completely covered by this set, and thus can be hidden from the outside without losing information.

Def 4.1 (cover number) *Let $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ be a hypergraph and $\mathcal{A} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\} \subseteq \mathcal{C}$. The* cover number *of $\mathcal{A}$, in symbols $\langle\!\langle \mathcal{A} \rangle\!\rangle$, is defined as the number of hyperedges covered by the subtrees of $\mathcal{A}$.*

$$\langle\!\langle \{\mathcal{T}_1, \ldots, \mathcal{T}_k\} \rangle\!\rangle \quad := \quad |\{\ell_e \,|\, e \in \mathcal{E},\ e \subseteq \mathit{leaves}(\mathcal{A}),\ \forall i.\ e \not\subseteq \mathit{leaves}(\mathcal{T}_i)\}|$$

Though this value tells us a lot about a candidate, it is itself not a good guideline. Recall that the set $\mathcal{C}$ has naturally always the highest possible cover number $|\mathcal{E}|$. We want to relate the cover number to the *size* of a candidate. Have a look at Figure 7. Which one of the choices is preferable?



Two clusterings of K_3 Two clusterings of a 6-ring

Figure 7: Considerations about the strength of a connection.

In the case of K_3, it is not intuitive, why one pair $\{A, B\}$ should be on a level below in the hierarchy. Rather we would like the left option to be taken. When considering the 6-ring instead, the six hyperedges are too weak an argument to group this big structure together in one go; we would favor the right alternative. So, three components with three links should be stronger than two with just one. But at the same time, two components with one link should be rated higher than six components with six links. This suggests, that size is not to be taken as a linear factor. Instead rather think of the number of possible connections in a set of size n, which is $\binom{n}{2} = \mathcal{O}(n^2)$. We propose the following rating function:

$$\mathbf{r}_{pref}(\mathcal{A}) \quad := \quad \frac{\langle\!\langle \mathcal{A} \rangle\!\rangle}{|\mathcal{A}|^2}$$

Comparing this to the examples in Figure 7, we can verify that $\mathbf{r}_{pref}$ precisely selects the options we argued for. In the following we refine $\mathbf{r}_{pref}$ by adding more structural information.

Def 4.2 (touch of a candidate) *Let $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ be a hypergraph and $\mathcal{F}$ be a forest over leaves $\mathcal{C}$. Then the* touch *of $\mathcal{A} \subseteq \mathcal{F}$ is defined as the labels from hyperedges that connect $\mathcal{A}$ with the rest of $\mathcal{H}$.*

$$touch(\mathcal{A}) \ := \ \{\, \ell_e \,|\, e \not\subseteq \, leaves(\mathcal{A}) \,\}$$

Def 4.3 (depth of a candidate) *The* depth *of a tree with only one node equals 0. Let $\mathcal{F}$ be a forest, $\mathcal{A} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\} \subseteq \mathcal{F}$. The* depth *of $\mathcal{A}$ is defined as*

$$depth\,(\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}) \quad := \quad 1 + \max_{1 \le i \le k} depth(\mathcal{T}_i)$$

It is intuitive that preference should be given to candidates with small depth. We might also be interested in the number of links, that *cannot* be hidden, i.e., the touch. If this number is big, the candidate is less desirable. Hence we propose the following improved rating function.

$$\mathbf{r}^{+}_{pref}(\mathcal{A}) := \frac{\langle\!\langle \mathcal{A} \rangle\!\rangle}{|\mathcal{A}|^2} + \frac{\varepsilon_1}{|touch(\mathcal{A})|} + \frac{\varepsilon_2}{depth(\mathcal{A})}$$

The parameters ε_1 and ε_2 are supposed to be chosen small and positive. For the experiments in section 5.1, the assignments $\varepsilon_1 := 1/1000$, $\varepsilon_2 := 1/100000$ were used.

Restricting the set of considered candidates In our formulation of the algorithm *partition_incrementally* we remained unclear what the considered candidates are. We want to weed out hopeless candidates, e.g., those not sharing any labels, before adding them to our priority queue. In a positive formulation, consider only candidates, that are extensions of interesting pairs.

Def 4.4 (interesting pair) *Given a hypergraph $\mathcal{H}(\mathcal{C}, \mathcal{E})$ and a forest $\mathcal{F}$ over leaves $\mathcal{C}$. An* interesting pair $\{\mathcal{T}_1, \mathcal{T}_2\}$ *is a subset of $\mathcal{F}$, such that touch$(\mathcal{T}_1) \cap$ touch$(\mathcal{T}_2) \neq \emptyset$.*

It is clear that every candidate that is *not* a superset of an interesting pair has cover number 0 and thus can be neglected. As it turns out in our implementation, the expensive part of the algorithm is the computation of the cover number. First computing interesting pairs and then extending them to candidates is an advantage.

The number of candidates can still be excessive, just consider a hyperedge connecting all vertices. Since the number of subsets of $\mathcal{C}$ is exponential in $|\mathcal{C}|$, an exhaustive enumeration is not feasible for large systems. If conservative techniques (like considering just extensions of interesting pairs) do not suffice, we have to apply a more rigorous pruning, even for the price of thereby ignoring good candidates. An obvious suggestion is to consider only candidates up to a certain size k, thus establishing an upper bound of $n^{k+1} - n - 1$ candidates. This k can be adjusted according to n, which provides a simple and reasonable method to prune the search.

5 Experimental Results

We implemented the algorithm from section 4.2 in an experimental version of the MOCHA verification tool [jmo00] (a predecessor was implemented in C, see [AHMQ98]). We extended the recent Java implementation, which make use of native libraries for symbolic model checking. Since our experiments only employ the enumerative check, given run-times and memory requirements are those of the Java Virtual Machine, executing on a Sun Enterprise 450 with UltraSPARC-II processors, 300 MHz. Together with an optimization in the enumerative check called *"Next" heuristic* [AW99], we are able to corroborate effectiveness and usability of our algorithm in some simple examples. We consider an asynchronous parity computer, a leader election protocol, and an opinion poll protocol.

5.1 Asynchronous Parity Computer

This example models a parity computer, designed as a binary tree (see Figure 8). The leaf nodes are *Client* modules, communicating a binary value to the next higher *Join*. A simple hand-shake protocol is devised by the two variables *ack* and *req*. All components are supposed to move asynchronously. Thus the join nodes have to wait for both values to be present, before reporting their exclusive-or upwards. The *Root* component, after receiving the result of the computation, hands down an acknowledgment. After this reaches a client, it is able to devise a fresh value. Thus 'computation' here is an ongoing process rather than a function terminated at some distinguished point.

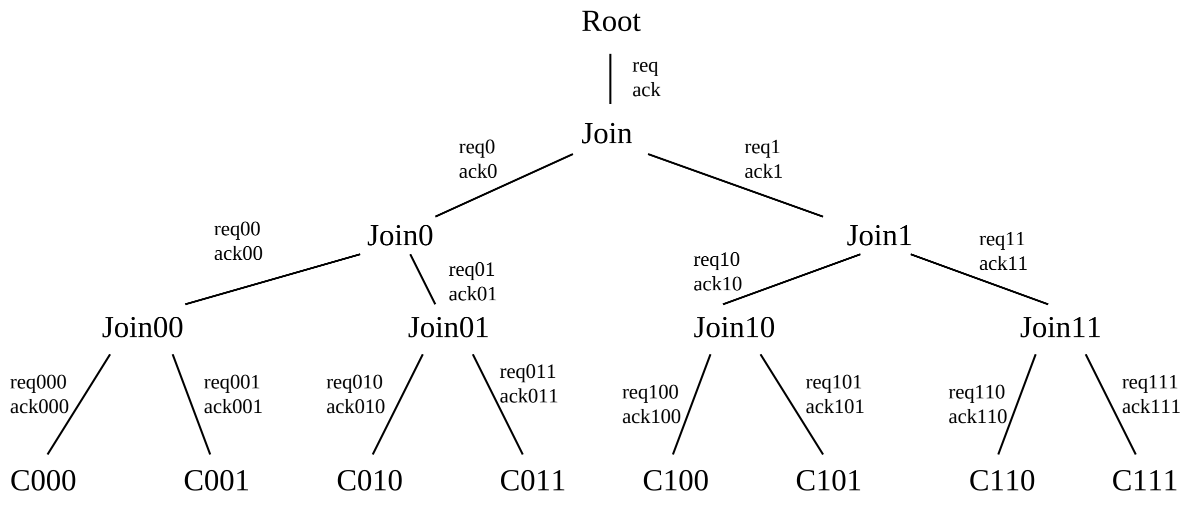


Figure 8: Layout of an asynchronous parity computer with eight clients.

We consider binary trees with n client nodes, where n varies from 3 to 8. The number of variables increases linearly with n, whereas the state-space grows exponentially. The sample question we pose is whether the module *Root* will ever output a value *zero* or *one*. We expect our model checking algorithm to falsify the claim, that it never will. The MOCHA specification for the case $n = 3$ is given in Appendix B.

It is obvious that the variables used for communication between clients and joins (like $req000$, $ack000$) do not contribute to this query. Rather, the behavior of some sub-tree rooted in a join can be simulated by replacing it with an appropriate client node. In an attempt to make the check more efficient, the local states of clients need not to be considered. It suffices to establish the reachability for a state, where some *req* is sent. This concept can be built into the enumerative model check algorithm, as described in detail in [AW99]. However, it requires an explicit structuring of the system: Sub-components have to be grouped together, irrelevant variables have to be *hidden* explicitly.

For a human user—understanding the structure and the problem at hand—it is intuitive to introduce a structuring bottom up. At a glance, he would spot the client modules to be leaves. However, for an automated algorithm, it presents a challenge. The difference between $\{Join1,\ Join11\}$ and $\{Client000,\ Join00\}$ is minor: Both pairs cover exactly two variables. Thus an incautious technique easily runs into errands. Figure 9 shows the results of automated hierarchical partitioning when using $\mathbf{r}_{pref}$ as rating function. Though it succeeds in incrementally hiding more and more variables, the obtained structure is uncomfortably deep.

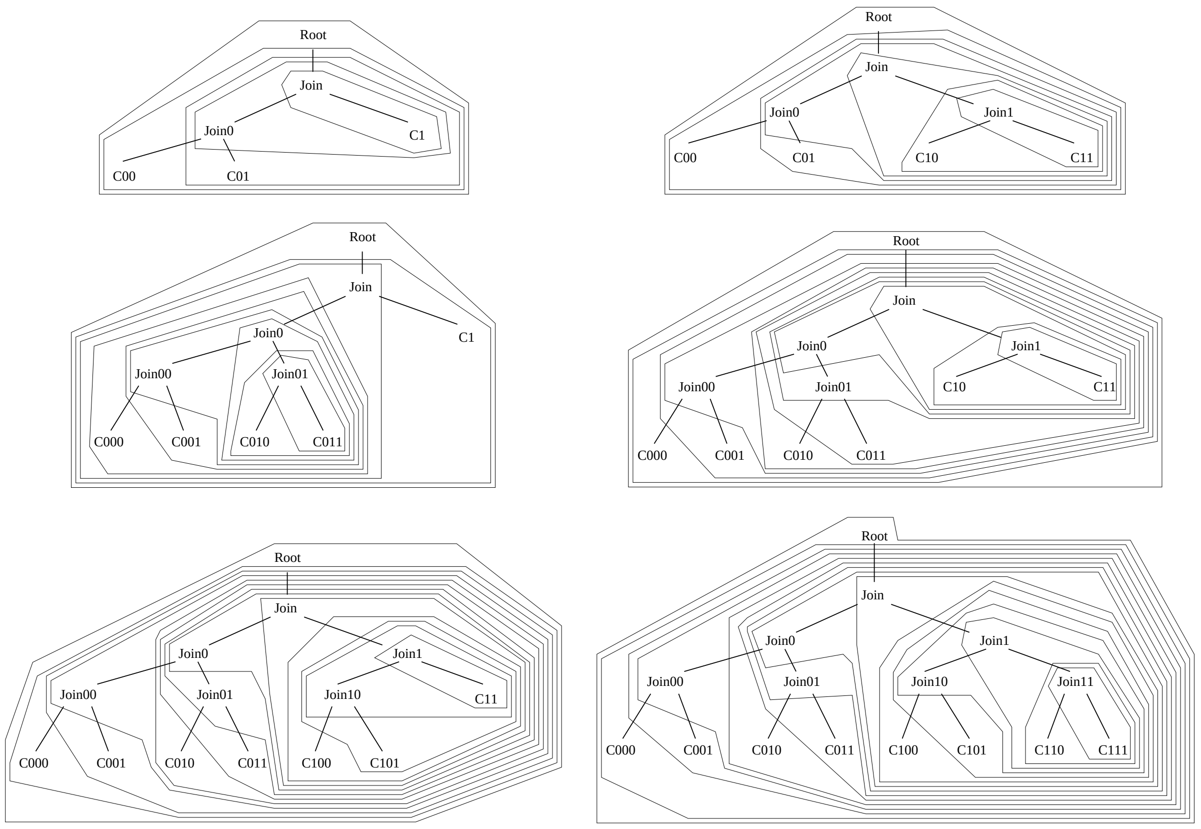


Figure 9: Applying $\mathbf{r}_{pref}$ on parity computers with three to eight clients.

The somewhat more sophisticated rating function $\mathbf{r}^{+}_{pref}$ performs far better, as seen in Figure 10. The parameters ε_1 and ε_2 were calibrated to $\varepsilon_1 := 1/1000$ and $\varepsilon_2 := 1/100000$, giving shallow structures a bigger bonus than those encompassing only few variables.

Table 2 allows us to verify, that the intuitively more adequate structure indeed leads to a better run-time performance with respect to an enumerative check. With “partition” we denote the preprocessing time used by *partition_incrementally* and “check” corresponds to the run-time of the model checking algorithm. All run-times are in milliseconds. We see that the time for computing the partitioning exceeds the model checking time for bigger examples. This explains by two facts: First, we never restricted the size of candidates and thus considered a number exponential in n. Second, the property we checked did not hold, thus the model checking algorithm was able to abort without having to explore the entire state space.

5.2 Leader Election in a Ring

We consider a leader election protocol as a second sample problem. The modules are arranged in a ring topology, where each buffered channel is modeled by a separate module. The

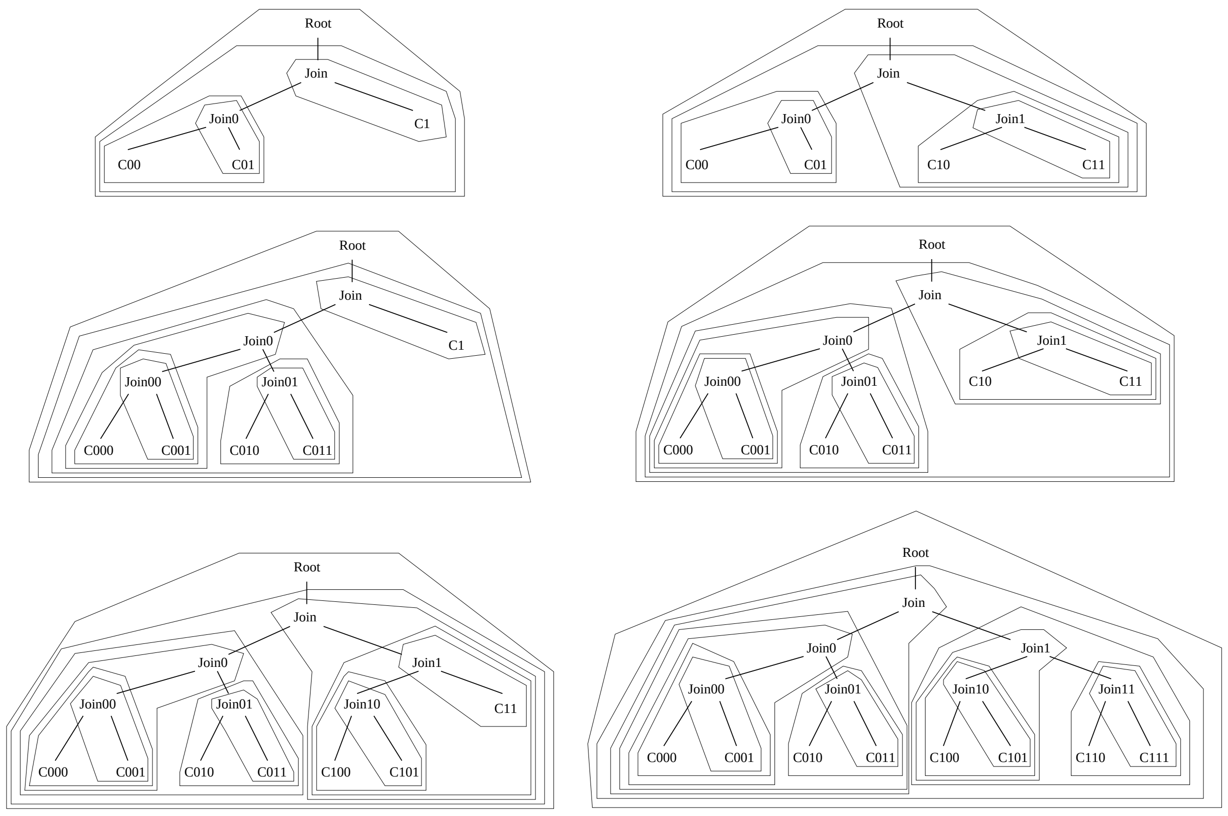


Figure 10: Using $\mathbf{r}^{+}_{pref}$ yields an intuitive hierarchical partitioning.

modules use a standard leader election protocol to elect the one with the highest (unique) id number: Every cell proceeds sending the highest id number it has seen so far, starting with his own. If a cell receives its own id number, it declares itself to be the leader (see [Lyn96] for an exhaustive treatment of this problem). Figure 11 shows how a ring with 3 cells is partitioned incrementally with application of rating function $\mathbf{r}^{+}_{pref}$. The checked property is a valid invariant: At no time there is more than one process denoted leader (a safety property). The gap in time performance between unstructured and clustered system was not extreme, but noticeable (Table 3). Unfortunately, we are not able to perform checks with larger rings, since the modeling of the links as finite state automata turns out to be very consumptive with respect to the state space. We included the Mocha specification in Appendix C.

5.3 Opinion Poll Protocol

The last sample problem is rather artificial. It is meant to demonstrate the behavior of our heuristic in a setting where there is no obvious choice.

size	partition	|hash table|	check
3	3˙227	97	556
4	4˙683	647	3˙507
5	6˙214	1˙945	11˙442
6	9˙314	16˙047	102˙920
7	19˙064	58˙353	433˙828
8	69˙006	[Memory out]	–

Applying $\mathbf{r}_{pref}$ as heuristic function

size	partition	|hash table|	check
3	134	53	349
4	313	119	787
5	712	141	1˙146
6	2˙742	207	1˙813
7	12˙804	273	2˙632
8	63˙834	471	4˙973

Applying $\mathbf{r}^{+}_{pref}$ with $\varepsilon_1 := \frac{1}{1000}$, $\varepsilon_2 := \frac{1}{100000}$

Table 2: Parity computer: Run-time performance of two heuristic functions.

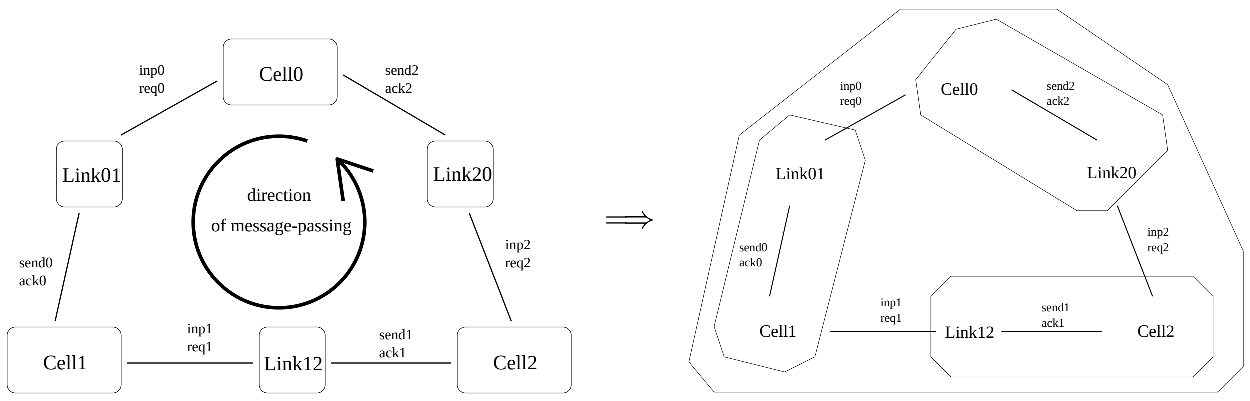


Figure 11: Leader election protocol with 3 cells, clustered via $\mathbf{r}^{+}_{pref}$.

size	|hash table|	model checking
2	563	6˙270
3	70˙797	1˙327˙756

Checking without partitioning

size	partition	|hash table|	model checking
2	217	563	4˙615
3	279	61˙455	661˙275

Applying $\mathbf{r}^{+}_{pref}$ before checking

Table 3: Checking leader election protocols without and with partitioning.

Consider a poll for a public opinion. There is a line of N pollers P_i and two non-connected lines of citizen A_i and B_i, plus two special citizen C and D. Poller P_1 starts raising an issue with a `Yes/No` question. Let us assume that the way one asks influences the answer. Poller P_1 starts of with an opinion he got from a random source (called $Master$). Poller P_{i+1} is influenced by P_i. The citizen are influenced by the pollers who asked and one other citizen. For example, A_{i+1} is influenced by A_i's opinion and A_1 by a random source N_A. In Figure 12 you see the communication pattern for the cases $N = 1$ and $N = 2$. The Mocha specification for the case $N = 1$ is found in Appendix D.

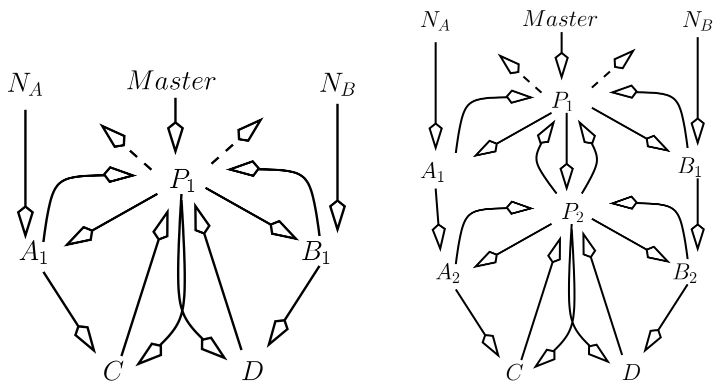


Figure 12: Opinion poll communication pattern for $N = 1$ and $N = 2$.

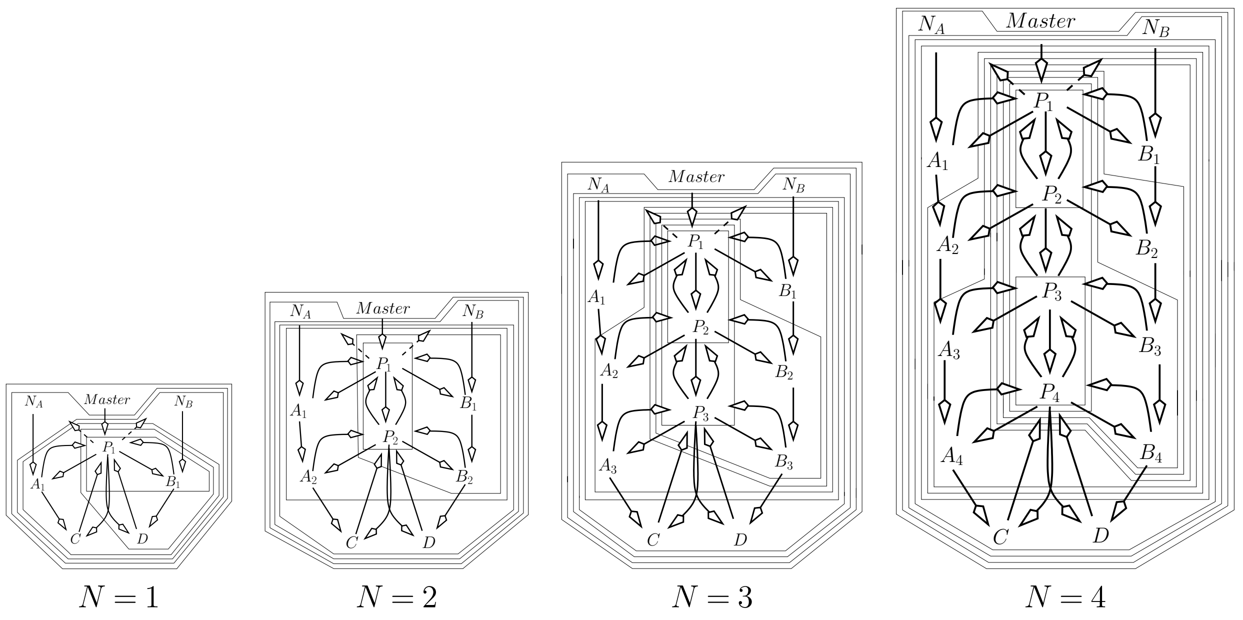


Figure 13: Opinion poll: Partitioned according to rating function $\mathbf{r}^+_{pref}$.

For $N = 1, 2, 3, 4$ we considered three invariants: *(i)* a false property that is easy to falsify, *(ii)* a false property that requires a special scenario (called *bad property* in the following), and *(iii)* a true property.

The experiments compare plain enumerative model-checking and application of the next heuristic, where the preprocessing follows one of the following strategies. *a. 2-merge*: Group any pair with a connection, i.e., follow Strategy I, *b. pref*: Partition incrementally according to rating function $\mathbf{r}_{pref}$, and *c. pref+size*: Partition incrementally according to rating function $\mathbf{r}^+_{pref}$. For the latter, we included the results of the preprocessing in Figure 13. It is interesting to note that sometimes triples were preferred to pairs. The quantitative comparison is listed in Table 4, run-times are in milliseconds. For the false properties *(i)* and *(ii)* the enumerative check is *slower* when sophisticated heuristics are applied. That explains by the fact, that it is more tedious to reach a counter-example scenario here. For the true property *(iii)*, the enumerative check improves with application of the next heuristic. For larger N the more sophisticated clustering techniques *pref* and *pref+size* perform slightly better than *2-merge*.

(i) **false Judgment:** `System |= (result = DontKnow)`							
N\Method	plain	2-merge		pref		pref+size	
1	742	2	(partition)	280	(partition)	274	(partition)
		854	(check)	754	(check)	861	(check)
2	3˙713	32	(partition)	1˙808	(partition)	1˙886	(partition)
		4˙313	(check)	6˙862	(check)	6˙850	(check)
3	32˙181	7	(partition)	1˙790	(partition)	2˙047	(partition)
		26˙330	(check)	87˙708	(check)	88˙879	(check)
4	345˙071	22	(partition)	5˙256	(partition)	4˙828	(partition)
		435˙529	(check)	1˙390˙739	(check)	1˙351˙527	(check)

(ii) **bad Judgment:** `System |= NoNegativeResult`							
N\Method	plain	2-merge		pref		pref+size	
1	1˙113	2	(partition)	238	(partition)	203	(partition)
		916	(check)	766	(check)	886	(check)
2	4˙846	5	(partition)	1˙625	(partition)	1˙667	(partition)
		3˙930	(check)	7˙130	(check)	6˙561	(check)
3	32˙580	7	(partition)	1˙788	(partition)	1˙920	(partition)
		29˙324	(check)	87˙827	(check)	73˙350	(check)
4	385˙951	20	(partition)	5˙476	(partition)	6˙458	(partition)
		375˙977	(check)	1˙665˙765	(check)	1˙306˙961	(check)

(iii) **true Judgment:** `System |= ~((result = DontKnow) & (result = Yes))`							
N\Method	plain	2-merge		pref		pref+size	
1	30˙565	2	(part.)	290	(part.)	292	(part.)
		23˙689	(check)	24˙369	(check)	24˙423	(check)
2	610˙131	5	(part.)	1˙787	(part.)	2˙148	(part.)
		454˙089	(check)	482˙600	(check)	482˙214	(check)
3	8˙488˙532	17	(part.)	2˙301	(part.)	2˙357	(part.)
		6˙392˙536	(check)	5˙920˙255	(check)	5˙865˙170	(check)
4	93˙557˙192	23	(part.)	5˙733	(part.)	5˙068	(part.)
		60˙934˙073	(check)	57˙762˙294	(check)	57˙165˙981	(check)

Table 4: Opinion poll protocol: Run-time comparison of different preprocessing methods.

6 Critique

Our proposed method gives no guarantee on how the obtained result compares to an optimal solution. Since we apply a variation of local search, it is to be expected that this method gets caught in local optima (for an overview on heuristic search techniques see e.g. [RN95]). Moreover, optimality in the sense of least cost does not imply minimal time- or space-consumption when running a model checking algorithm. Part of the difficulties with quantifying this algorithm lies in the structural complexity of the problem itself: It is hard to capture the characteristics of a system within a single number.

Our case studies teach the lesson that—when using an incremental approach— both size and depth of the candidates have to be taken into account. It remains unclear, how this should be reflected in general. The values for ε_1 and ε_2 were chosen according to fit the example of the parity computer. It would be desirable to investigate the impact of parameter changes in general, but we lack apt mathematical means to do so.

Considered Candidates and Rating Functions There are several improvements that can be taken into account. For example, the number of considered candidates in each step should be further restricted. The interesting pairs technique provided a first step, but it would suffice to consider only supersets of present hyperedges.

If the number of vertices is large, we have to apply more rigorous pruning. We proposed a simple restriction on the size of considered candidates. There is limited potential that more subtle techniques yield better results in practice.

Unfortunately, the best upper bound on the number of candidates is exponential in the number of components. Simple syntactic restrictions do not change this situation. Assume the size of hyperedges is bounded by a constant. This does not suffice to derive an upper bound on the size of reasonable candidates. Consider a ring of n components $A_0, \ldots, A_{n-1}$, where every pair $(A_i,\ A_{i+1 \text{ MOD } n})$ is connected via multiple hyperedges. Though all the hyperedges are of size 2, the candidate $\{A_0, \ldots, A_{n-1}\}$ cannot be neglected.

Unfortunately, we cannot expect efficient techniques to guarantee preservation of the best candidate: For some simple rating functions, the computational complexity of finding the the best candidate is high. In particular, we can encode the Max Clique problem[5] by using this rating function:

$$\mathbf{r}_\varepsilon(\mathcal{A}) \ := \ \frac{|\mathcal{A}|}{\varepsilon + |\mathcal{A}| \cdot (|\mathcal{A}| - 1) - \langle\!\langle \mathcal{A} \rangle\!\rangle}$$

If the input is a simple graph and ε is chosen sufficiently small, the largest clique is precisely the candidate with the highest rating. That means that in every iteration we have to solve a *NP*-hard problem.

Note that this does not imply, that finding the best candidate is hard for *every* rating function. If we choose $\mathbf{r}(\mathcal{A}) := \langle\!\langle \mathcal{A} \rangle\!\rangle$, then the single best candidate is always the complete set $\mathcal{C}$. We were not able to establish the hardness for interesting classes of rating functions. But we believe that there is little hope to get this step more than reasonably efficient.

Outlook Our method is not limited to Mocha. It can be applied in other settings where connected entities have to be structured or re-structured. The parameters can be adjusted

[5]See e.g. [GJ79], listed under graph theory, no.19.

accordingly. We believe that the obtained structure can be exploited by other means, e.g. by building abstractions based on this particular hierarchy.

In our considered application, the difficulty of the model checking problem relies on the size of the state space. This is typically exponential in the number of modules. In many cases, the size of the state space turns out to be the bottleneck, that can be overcome only by application of various heuristics. Hence, automated tree-indexing can be considered a good investment.

Related Work Hierarchical structures find a wide range of application in design, description and physical organization of both software and hardware.

In particular, the decomposition of large circuits in VLSI layout turns out to be a crucial problem and has received a respectable amount of attention. Here the partitions are typically shallow (i.e., of depth two) and mainly motivated by size constraints that single components have to meet. Optimality is typically described as the least number of components with as few as possibly connections.

Similar structures called classification trees (e.g. [GRD91]) are used as expressive decision trees over large sets of data. The internal nodes are labeled by distinguishing criteria and all leaf nodes are distinguishable. Finding expressive classification trees is computationally hard.

Though various advanced techniques have been developed for these problems, to the best of our knowledge none of them is applicable in the considered case. In our setting, *every* tree-indexing is a feasible solution, there is no constraint satisfaction component and there might well be two leaves that are alike. Moreover, our notion of optimality is less clearly shaped. In the particular application in model checking, we want to minimize the time and memory requirements of our model checking algorithm. Hopes are low, that we can practicably evaluate this characteristic. Thus we have to rely on guidelines.

Open Questions We showed that finding an optimal solution with respect to our cost function is *NP*-hard, but this does not preclude the existence of an polynomial approximation scheme. Also, it remains unknown, how the computational complexity compares with respect to other cost functions, like $\mathit{depth\_cost}(\mathcal{T}) := \mathit{depth}(\mathcal{T})$ or $\mathit{cost}(\mathcal{T}) := \sum_e |\mathit{leaves}(\mathcal{T}_e)|$. It is conjectured that the tree-indexing problem remains *NP*-complete in both cases.

7 Summary

We developed a notion of optimal hierarchical partitioning of a hypergraph and proved the problem to be *NP*-complete. We presented a simple and scalable method to automate hierarchical partitioning. With this greedy approach we proposed a feasible strategy to battle the combinatorial and computational complexity of the problem. We argued that—in order to achieve a good result—a candidate for grouping together should be evaluated with respect to four criteria: Number of covered hyperedges, size, number of occurring hyperedges, and structural depth.

We implemented our algorithm and validated its performance on small and medium sized examples. It is considered to include this method in the MOCHA model checking tool as an option.

Acknowledgments: We thank Bow-Yaw Wang and Radu Grosu for many useful comments and invaluable help on MOCHA implementation details. This research was supported in part by SRC award 99-TJ-688.

References

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[AHMQ98] R. Alur, T. A. Henzinger, F. Y. C. Mang, and S. Qadeer. MOCHA: Modularity in model checking. *Lecture Notes in Computer Science*, 1427:521–525, 1998.

[AW99] Rajeev Alur and Bow-Yaw Wang. "Next" Heuristic for On-the-fly Model Checking. In *Proceedings of the Tenth International Conference on Concurrency Theory (CONCUR'99)*, LNCS 1664, pages 98–113. Springer-Verlag, 1999.

[Ber73] Claude Berge. *Graphs and Hypergraphs.* Number 6 in North-Holland Mathematical Library. North-Holland Publ. Co., American Elsevier Publ. Co. Inc., Amsterdam, 1973.

[CE82] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons from branching time temporal logic. *Lecture Notes Comp. Sci.*, 131:52–71, 1982.

[CK96] Edmund M. Clarke, Jr. and Robert, P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 6(33):61–67, June 1996.

[CPS91] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. Technical Report 91-24, Technical University of Aachen (RWTH Aachen), 1991.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[Fla88] Phillippe Flajolet. Mathematical Methods in the Analysis of Algorithms and Data Structures. Lecture Notes for *A Graduate Course on Computation Theory*, Udine (Italy), Fall 1984. In Egon Börger, editor, *Trends in Theoretical Computer Science*, pages 225–304. Computer Science Press, 1988.

[Fla97] Philippe Flajolet. A problem in Statistical Classification Theory, 1997. `http://pauillac.inria.fr/algo/libraries/autocomb/schroeder-html/schroeder1.html`.

[FZV94] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132:1–35, 1994. A preliminary version is available in INRIA Research Report RR-1830.

[GJ79] Michael R Garey and David S Johnson. *Computers and Intractibility, a Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[GJS76] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976.

[GRD91] Saul B. Gelfand, C. S. Ravishankar, and Edward J. Delp. An iterative growing and pruning algorithm for classification tree design. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-13(2):163–174, February 1991.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[Hol97] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[jmo00] Mocha: Exploiting Modularity in Model Checking, 2000. see `http://www.cis.upenn.edu/~mocha`.

[KST93] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity.* Birkhäuser, 1993.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[Mäk89] Erkki Mäkinen. How to draw a hypergraph. Technical Report A-1989-3, University of Tampere, Department of Computer Science, 1989.

[RN95] S. J. Russell and P. Norvig. *Artificial Intelligence. A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Sch70] Ernst Schröder. Vier combinatorische probleme. *Zentralblatt. f. Math. Phys.*, 15:361–376, 1870.

A Edge-Guided Tree-Indexing is *NP*-complete

Theorem 1 *For a multi graph $G = (V, E)$ and an integer K, deciding whether there exists a tree-indexing $\mathcal{T}$ with cost at most K is NP-complete.*

Proof. Containment in *NP* is simple, for we can guess any possible tree-indexing $\mathcal{T}$ and compute $cost(\mathcal{T})$ in polynomial time. We show *NP*-hardness by reduction from the following *NP*-complete problem.

Minimum Cut Into Equal-Sized Subsets **[GJS76]**[6] Given a graph $G = (V, E)$ with specified vertices $s, t \in V$ and a positive integer K. It is *NP*-complete to answer the following question. Is there a partition of V into disjoint sets V_1, V_2 such that $s \in V_1$, $t \in V_2$, $|V_1| = |V_2|$, such that the number of edges with one endpoint in V_1 and the other endpoint in V_2 is no more than K?

Note that $n := |V|$ is restricted to be even. For $n = 2$ there exists only one solution, thus we consider $n \geq 4$. Furthermore, we assume that that $m := |E| > n/4$. For smaller m the problem is trivial, since we have at least $n/2$ isolated vertices.

Reduction. For every instance (G, s, t, K) we construct—in polynomial time and logarithmic space—an instance (G', K'), such that there exists a partition V_1, V_2 with cost $\leq K$ if and only if there exists a tree-indexing $\mathcal{T}$ of G' with $cost(\mathcal{T}) \leq K'$.

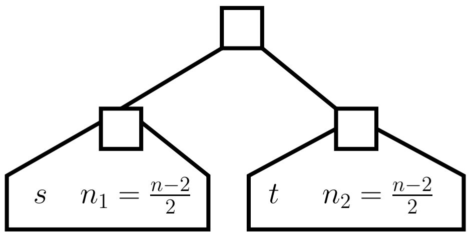


Figure 14: Binary, shallow, and balanced tree-indexing $\mathcal{T}^*$.

The idea is to augment G in a way, such that a tree-indexing with lowest cost has the shape of a balanced tree of depth 2 (see Figure 14). To achieve this, we add edges leading to the nodes s and t. We call s and t *attractors* in the following. The number of edges between V_1 and V_2 then corresponds to the number of edges between the subtrees of $\mathcal{T}$ plus some offset.

Given $G = (V, E), s, t, K$, $|V| = n$, $|E| = m$. Let $V' := V \setminus \{s, t\}$. Then we define G' to be the graph with vertices V and $4m^2$ additional edges between each $v \in V'$ and every

[6]See also: [GJ79], comment to problem Minimum Cut into Bounded Sets (ND17).

attractor. We call these edges β-edges and use $\beta := 4\,m^2$ as a parameter.

$$\begin{array}{lll} cost^{+} & := & \beta\ (2\,(n/2-1)\ \cdot\ 2\,n/2\ +\ (n-2)\ \cdot\ 2\,n) \\ K' & := & cost^{+}\ +\ (m-K)\ \cdot\ 2\,n/2\ \ +\ \ K\ \cdot\ 2\,n \end{array}$$

It suffices to show that an optimal tree-indexing has the shape of a balanced tree of depth 2. Then the cost for the newly introduced edges is fixed ($cost^{+}$). We still have to pay for the edges that originated from E. If they live inside a subtree, they are cheap ($2\,n/2$), else they are punished with factor $2\,n$. It is clear that cost K' can be achieved if and only if a balanced partition of V with at most K edges between V_1 and V_2 is possible.

Let $\mathcal{T}^*$ be a tree-indexing like in Figure 14. In the worst case, all m original edges in E are in between the subtrees. Thus we can give an upper bound $cost^*$ on $cost(\mathcal{T}^*)$.

$$\begin{array}{lll} cost^{*} & := & \beta\ (2\,(n/2-1)\ \cdot\ 2\,n/2\ +\ (n-2)\ \cdot\ 2\,n)\ \ \ +\ \ m\ \cdot\ 2\,n \\ & = & 3\,\beta\,n(n-2)+2\,mn \end{array}$$

Lemma 1 *Let $\mathcal{T}$ be a tree-indexing of depth 1. Then $cost(\mathcal{T})\ >\ cost^*$.*

Proof. The cost of $\mathcal{T}$ is precisely

$$\begin{array}{lll} (2\,\beta\,(n-2)\ +\ m)\ \cdot\ 2\,n & = & \\ 4\beta\,n(n-2)+2\,mn & > & 3\beta\,n(n-2)+2\,mn. \end{array}$$

///

Lemma 2 *Let $\mathcal{T}$ be a tree-indexing, where attractors s and t live in the same subtree. Then $cost(\mathcal{T})\ >\ cost^*$.*

Proof. Suppose a tree-indexing $\mathcal{T}$ where t and s live in the same subtree. To estimate the cost of $\mathcal{T}$, we can assume that the tree containing s, t has no siblings within subtree 1 (there is additional punishment by increased depth and no gain). The same holds true for vertices located in a subtree starting at the level of s and t. Thus we can assume that the two attractors share a subtree of depth 1 with n_1 additional vertices and that the remaining $(n-n_1-2)$ graph nodes are collected in a second subtree, for in alternative structures the relevant costs are the same or worse. Thus the only scenario to consider is displayed in Figure 15.

A lower bound on the cost of structure $\mathcal{T}$ is obtained by looking only at β edges. The costs for subtree 1 is $\beta\,(2\,n_1)\cdot 2\ (2+n_1)$ and the graph violations amount to cost $2\,\beta\ (n-n_1-2)\cdot 2\,n$.

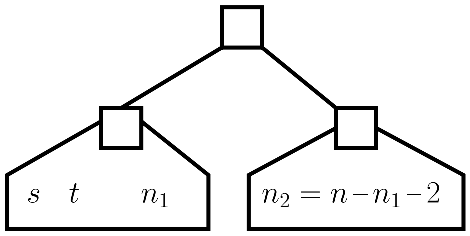


Figure 15: Best case of a tree-indexing $\mathcal{T}$ with attractors in the same subtree.

Comparing the costs yields

$$\begin{aligned}
cost(\mathcal{T}) - cost^* &= \\
\beta\,(2\,n_1) \cdot 2\;(2 + n_1) + 2\,\beta\;(n - n_1 - 2) \cdot 2\,n - (3\,\beta\,n(n-2) + 2\,m\,n) &= \\
8\,\beta\,n_1 + 4\,\beta\,{n_1}^2 + \beta\,n^2 - 4\,\beta\,nn_1 - 2\,\beta\,n - 2\,mn &= \\
\beta\;(n^2 - 4\,n + 4 - 4\,nn_1 + 8\,n_1 + 4\,n_1^2) + \beta\,(2\,n - 4) - 2\,mn &= \\
\beta\,((n-2) - 2\,n_1)^2 + \beta\,(2\,n - 4) - 2\,mn &\geq \\
4\beta - 2\,mn &\geq \\
4\,mn - 2\,mn &> 0
\end{aligned}$$

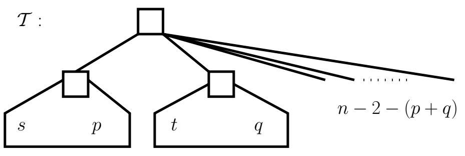
///

We can assume now, that the optimal solution (i.e. this that allows the largest K' and thus the largest K) is non-monolithic and that attractors s and t live in different subtrees. Before we argue that more than two subtrees are too expensive, we need to state a lower bound on the cost of subtrees with one attractor.

Lemma 3 *Let $\mathcal{T}$ be a tree-indexing where the root has $\geq$ 3 children, attractor s lives in subtree 1 and attractor t in subtree 2. Then $cost(\mathcal{T})\ >\ cost^*$.*

Proof. Assume that subtrees 1 and 2 contain, in addition to the attractors, p respectively q nodes, $p + q < n - 2$. Make a case split on the depth of the tree-indexing.

(i) The depth of the tree-indexing is 2.
Then the cost can be estimated as in Figure 16. It suffices to show the following



$cost(\mathcal{T}) \geq$

$\beta\,p \cdot 2\,(p+1)$	inside subtree 1
$\beta\,q \cdot 2\,(q+1)$	inside subtree 2
$\beta\,n \cdot 2\,n$	to attractor in other subtree
$\beta\,(n\text{-}2\text{-}p\text{-}q) \cdot 2\,n$	additional rightmost violations

Figure 16: Lower bound on cost of a non-binary tree-indexing $\mathcal{T}$.

inequality:

$$\begin{array}{rrcl}
& \beta p \cdot 2(p+1) + \beta q \cdot 2(q+1) + \beta n \cdot 2n + \beta(n\text{-}2\text{-}p\text{-}q) \cdot 2n & > & cost^* \\
\Leftrightarrow & \beta\,(n^2 - 2n - 2np - 2nq + 2p + 2p^2 + 2q + 2q^2) - 2mn & > & 0 \\
\Leftrightarrow & \beta \begin{pmatrix} n^2 - 4n + 4 - 2np - 2nq + p^2 + q^2 + 2pq + 4p + 4q \\ +2n - 4 \qquad\qquad +p^2 + q^2 - 2pq - 2p - 2q \end{pmatrix} - 2mn & > & 0 \\
\Leftrightarrow & \beta(\underbrace{((n-2)-(p+q))^2}_{\geq 0} + \underbrace{(p-q)^2}_{\geq 0} + 2\underbrace{(n-p-q-2)}_{\geq 1}) - 2mn & > & 0 \\
\Leftarrow & 4m^2 \cdot 2 & > & 2mn
\end{array}$$

(ii) The depth of the tree-indexing is at least 3.
Then there are $(n-2-p-q)+(n-2) \geq n-1$ edges that have to be payed for with factor $\beta \cdot 3n$.

$$\begin{array}{rcl}
3\beta n(n-1) \;\; - cost^* & = & \\
3\beta n \;\; - 2mn & > & 0
\end{array}$$

///

Assuming that the root has exactly two children, we can now exclude tree-indexings of unnecessary depth.

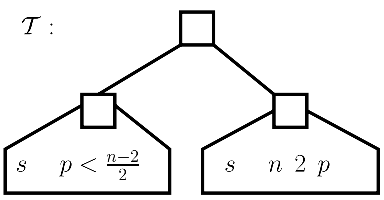
Lemma 4 *Let $\mathcal{T}$ be a tree-indexing where the root has 2 children, in each subtree lives one attractor and depth($\mathcal{T}$) ≥ 3. Then cost($\mathcal{T}$) $>$ cost*.*

Proof. Every of the $n-2$ leaves in $V \setminus \{s,t\}$ has β edges to the attractor in the other subtree. It suffices to estimate the internal costs of the two subtrees with $\beta 2 \cdot 2$ (we have at least two edges somewhere inside).

$$\begin{array}{rlcl}
& \beta\,(4 \;+\; (n-2)\,3n) & > & cost^* \\
\Leftrightarrow & 4 \cdot 4m^2 & > & 2mn
\end{array}$$

///

Lemma 5 *Let $\mathcal{T}$ be a tree-indexing where the root has 2 children, in each subtree lives one attractor, depth($\mathcal{T}$) $= 2$ and subtree 1 contains $p < (n-2)/2$ nodes from $V \setminus \{s,t\}$. Then cost($\mathcal{T}$) $>$ cost*.*



$cost(\mathcal{T})$:

$\beta\, p \,\cdot\, 2\,(p+1)$	inside subtree 1
$\beta\,(n{-}2{-}p) \,\cdot\, 2\,(n{-}2{-}p)$	inside subtree 2
$\beta\,(n{-}2) \,\cdot\, 2\,n$	to attractor in other subtree

Figure 17: Lower bound on cost of an unbalanced tree-indexing $\mathcal{T}$.

Proof. We can estimate $cost(\mathcal{T})$ as in Figure 17.

$$\begin{array}{rlcl} & \beta\ (p\,\cdot\,2\,(p+1) + (n-2-p)\,\cdot\,2\,(n-2-p) + (n-2)\,\cdot\,2\,n) & > & cost^* \\ \Leftrightarrow & \beta(\underbrace{4\,p^2 + 8\,p + n^2 - 4\,n - 4\,np + 4}_{f_n(p)}) & > & 2\,mn \end{array}$$

The term in $f_n(p)$ is minimal for $\frac{d}{dp}\,f_n(p) = 0$, i.e. for $p = n/2 - 1$. Since $p \leq n/2 - 2$ and f_n is quadratic in p, it suffices to check the value $p_0 = n/2 - 2$.

$$\begin{array}{rcl} f_n(n/2-2) & = & 2\ (1/2\,n - 2)\,(-1 + 1/2\,n) + n\,(1/2\,n + 1) - (n-2)\,n \\ & = & 4 \end{array}$$

Since $4\,\beta = 16\,m^2\ >\ 4\,mn\ >\ 2\,mn$, this proves the lemma.

///

Thus we can be sure that only binary, shallow and balanced tree-indexings like in Figure 14 are candidates for optimal solutions. The soundness of the reduction follows.

□

B Parity Computer (three clients)

```
/* ****************************************
 *  Asynchronous Parity Computer
 * ****************************************
 */

type ReqType is { none, zero, one, ready }
type AckType is { idle, wait, null, pos }

module Root is
  interface ack : AckType;
  external req : ReqType;
  private loc : (0..3);
lazy atom root controls ack, loc reads req, loc
  init
    [] true -> ack' := idle; loc' := 0;
  update
    [] (loc = 0) & (req = zero) -> ack' := wait; loc' := 1;
    [] (loc = 0) & (req = one) -> ack' := wait; loc' := 2;
    [] (loc = 1) & (req = ready) -> ack' := null; loc' := 3;
    [] (loc = 2) & (req = ready) -> ack' := pos; loc' := 3;
    [] (loc = 3) & (req = none) -> ack' := idle; loc' := 0;
```

```
module Client is
   interface req : ReqType;
   external ack : AckType;
   private loc : (0..3);
lazy atom client controls req, loc reads ack, loc
  init
    [] true -> req' := none; loc' := 0;
  update
    [] (loc = 0) -> req' := zero; loc' := 1;
    [] (loc = 0) -> req' := one; loc' := 1;
    [] (loc = 1) & (ack = wait) -> req' := ready; loc' := 2;
    [] (loc = 2) & ((ack = pos) | (ack = null)) -> req' := none; loc' := 3;
    [] (loc = 3) & (ack = idle) -> loc' := 0;

module Join is
  interface req : ReqType; ack0, ack1: AckType;
  external req0, req1 : ReqType; ack: AckType;
  private loc : (0..7);
lazy atom join controls req, ack0, ack1, loc reads req0, req1, ack, loc
  init
    [] true -> req' := none; loc' := 0; ack0' := idle; ack1' := idle;
  update
    [] (loc = 0) & (req0 = zero) & (req1 = zero) -> ack0' := wait; ack1' := wait; loc' := 1;
    [] (loc = 0) & (req0 = zero) & (req1 = one) -> ack0' := wait; ack1' := wait; loc' := 2;
    [] (loc = 0) & (req0 = one) & (req1 = one) -> ack0' := wait; ack1' := wait; loc' := 1;
    [] (loc = 0) & (req0 = one) & (req1 = zero) -> ack0' := wait; ack1' := wait; loc' := 2;
    [] (loc = 1) & (req0 = ready) & (req1 = ready) -> req' := zero; loc' := 3;
    [] (loc = 2) & (req0 = ready) & (req1 = ready) -> req' := one; loc' := 3;
    [] (loc = 3) & (ack = wait) -> req' := ready; loc' := 4;
    [] (loc = 4) & (ack = null) -> req' := none; loc' := 5;
    [] (loc = 4) & (ack = pos) -> req' := none; loc' := 6;
    [] (loc = 5) & (ack = idle) -> ack0' := null; ack1' := null; loc' := 7;
    [] (loc = 6) & (ack = idle) -> ack0' := pos; ack1' := pos; loc' := 7;
    [] (loc = 7) & (req0 = none) & (req1 = none) -> ack0' := idle; ack1' := idle; loc' := 0;

/* ------------------------------------------------------------------- */

module C00 is Client[req,ack := req00,ack00]
module C01 is Client[req,ack := req01,ack01]
module C1  is Client[req,ack := req1,ack1]

module system3 is
  Root || Join || J0 || C00 || C01 || C1

/* ------------------------------------------------------------------- */

predicate notyield is ~(ack = null | ack = pos)

/* ------------------------------------------------------------------- */

judgment J3 is system3 |= notyield
```

C Leader Election Protocol (two processes)

```
/*  ***************************************
 *  Leader Election Protocol 2
 *  **************************************
 */

type ReqType is  { idle, high }
type StatusType is  { unknown, broken, chosen }

type AckType is  { null, ok }

module Cell is
        interface status: StatusType;
                  sendX:  ( 0 .. 4 );
                  req: ReqType;
                  loc:  ( 0 .. 5 );
                  u:  ( 0 .. 4 );
        external ack: AckType;
                 inp:  ( 0 .. 4 );
                 uid:  ( 0 .. 4 );

lazy atom A0 controls status, sendX, req, loc, u
             reads    status, sendX, req, loc, u, ack, inp, uid
     init
        [] true -> u' := 0; loc' := 0; req' := idle; status' := unknown; sendX' := 0;
     update
        [] loc = 0      -> u' := uid; loc' := 1;
        [] loc = 1      -> sendX' := u; loc' := 2;
        [] (loc = 2) & (ack = ok)       -> sendX' := 0; loc' := 3;
        [] loc = 3      -> loc' := 4; req' := high;
        [] (loc = 4) & (inp = uid)      -> req' := idle; status' := chosen; loc' := 5;
        [] (loc = 4) & (inp > uid)      -> req' := idle; u' := inp; loc' := 1;
        [] (loc = 4) & (inp > 0) & (inp < uid)  -> req' := idle; loc' := 3;

module Initialize is
        interface uid0, uid1: ( 0 .. 4 );

lazy atom A2 controls uid0, uid1
             reads    uid0, uid1
     init
        [] true -> uid0' := 1;
                   uid1' := 4;
     update

module Link is
        interface ack: AckType;
                  inp:  ( 0 .. 4 );
        external  sendX:  ( 0 .. 4 );
                  req: ReqType;
        private   B1, B2, B3:  ( 0 .. 4 );
                  items:  ( 0 .. 3 );
                  loc:  ( 0 .. 4 );

lazy atom A1 controls ack, inp, B1, B2, B3, items, loc
             reads    ack, inp, B1, B2, B3, items, loc, sendX, req
     init
        [] true -> loc' := 0; B1' := 0; B2' := 0; B3' := 0; items' := 0; ack' := null; inp' := 0;
     update
        [] (loc = 0) & (sendX > 0)      -> loc' := 1;
        [] (loc = 1) & (items = 0)      -> loc' := 2; items' := 1; ack' := ok; B1' := sendX;
        [] (loc = 1) & (items = 1)      -> loc' := 2; items' := 2; ack' := ok; B2' := sendX;
        [] (loc = 1) & (items = 2)      -> loc' := 2; items' := 3; ack' := ok; B3' := sendX;
        [] (loc = 2) & (sendX = 0)       -> loc' := 0; ack' := null;
        [] (loc = 0) & (req = high) & (items > 0)       -> loc' := 3;
        [] (loc = 3) & (items = 1)      -> inp' := B1; items' := 0; B1' := 0; loc' := 4;
        [] (loc = 3) & (items = 2)      -> inp' := B1; items' := 1; B1' := B2; B2' := 0; loc' := 4;
        [] (loc = 3) & (items = 3)      -> inp' := B1; items' := 2; B1' := B2; B2' := B3; B3' := 0; loc' := 4;
        [] (loc = 4) & (req = idle)     -> inp' := 0; loc' := 0;
```

```
module Cell0 is Cell [ status,  sendX,  req,  ack,  inp,  uid,  u,  loc :=
                       status0, sendX0, req0, ack0, inp0, uid0, u0, loc0 ]

module Cell1 is Cell [ status,  sendX,  req,  ack,  inp,  uid,  u,  loc :=
                       status1, sendX1, req1, ack1, inp1, uid1, u1, loc1 ]

module Link0_1 is Link [ ack,  inp,  sendX,  req :=
                         ack0, inp1, sendX0, req1 ]

module Link1_0 is Link [ ack,  inp,  sendX,  req :=
                         ack1, inp0, sendX1, req0 ]

module System is Initialize
        || Cell0
        || Cell1
        || Link0_1
        || Link1_0

/* ---------------------------------------------------------------------- */

predicate consistent is ~((status0 = chosen) & (status1 = chosen))

/* ---------------------------------------------------------------------- */

judgment J1 is System |= consistent
```

D Opinion Poll Protocol (one poller)

```
/*  ***************************************
 *  Opinion Poll 1
 *  ***************************************
 */

type opinionType is  { Yes, No, DontKnow }

type statusType is  { Init, Wait, Done }

/* ---------- Templates ---------- */

module Poller is
        interface tellA, tellB, tellDown, tellUp, tellUpEcho: opinionType;
        external hearA, hearB, hearUpA, hearUpB, master: opinionType;
        private opinion: opinionType; status: statusType;

lazy atom POLL controls opinion, tellA, tellB, tellDown, status, tellUp, tellUpEcho
               reads    hearA, hearB, hearUpA, hearUpB, status, opinion, master
     init
        [] true -> opinion' := DontKnow; status' := Init; tellA' := DontKnow;
                   tellB' := DontKnow; tellDown' := DontKnow;
                   tellUp' := DontKnow; tellUpEcho' := DontKnow;
     update
        [] (status = Init) & (opinion = DontKnow) ->  opinion' := master;
        [] (status = Init) & ((opinion = Yes) | (opinion = No)) ->
                   tellA' := opinion; tellB' := opinion; tellDown' := opinion; status' := Wait;
        [] status = Wait & (hearA = Yes) & (hearB = Yes) & (hearUpA = Yes) & (hearUpB = Yes) ->
                   opinion' := Yes; tellUp' := Yes; tellUpEcho' := Yes; status' := Done;
        [] status = Wait & (hearA = No) & (hearB = No) & (hearUpA = No) & (hearUpB = No) ->
                   opinion' := No; tellUp' := No; tellUpEcho' := No; status' := Done;
```

```
module Noise is
        interface tell: opinionType;
lazy atom FLIP controls tell
     init
        [] true -> tell' := Yes;
        [] true -> tell' := No;
     update


module Citizen is
        external hearUp, hearSide: opinionType;
        interface tellDown, tellUp: opinionType;
        private opinion: opinionType; status: statusType;

lazy atom VOTER controls tellDown, tellUp, status, opinion
                reads    hearUp, hearSide, status, opinion
     init
        [] true -> opinion' := Yes; status' := Init; tellDown' := DontKnow; tellUp' := DontKnow;
        [] true -> opinion' := No; status' := Init; tellDown' := DontKnow; tellUp' := DontKnow;
     update
        [] (status = Init) & (hearUp = Yes) & (hearSide = Yes)  ->
                   tellDown' := Yes; tellUp' := Yes; status' := Done;
        [] (status = Init) & (hearUp = No) & (hearSide = No)    ->
                   tellDown' := No; tellUp' := No; status' := Done;
        [] (status = Init) & (hearUp = Yes) & (hearSide = No)   ->
                   tellDown' := opinion; tellUp' := opinion; status' := Done;
        [] (status = Init) & (hearUp = No) & (hearSide = Yes)   ->
                   tellDown' := opinion; tellUp' := opinion; status' := Done;

/* ---------- Module Definitions ---------- */

module Master is Noise [ tell := hear_p_1_up ]
module P_1 is Poller [
  master, tellA, tellB, tellDown, hearA, hearB, hearUpA, hearUpB, tellUp, tellUpEcho :=
  hear_p_1_up, hear_a_1_side, hear_b_1_side, hear_cd, tell_a_1_up, tell_b_1_up, tell_c_up, tell_d_up,
                                                                 result, resultEcho ]
module N_A is Noise [ tell := hear_a_1_up ]
module N_B is Noise [ tell := hear_b_1_up ]
module A_1 is Citizen [ hearUp, hearSide, tellDown, tellUp :=
                        hear_a_1_up, hear_a_1_side, hear_c_side, tell_a_1_up ]
module B_1 is Citizen [ hearUp, hearSide, tellDown, tellUp :=
                        hear_b_1_up, hear_b_1_side, hear_d_side, tell_b_1_up ]
module C is Citizen [ hearUp, hearSide, tellDown, tellUp := hear_cd, hear_c_side, desert_c, tell_c_up ]
module D is Citizen [ hearUp, hearSide, tellDown, tellUp := hear_cd, hear_d_side, desert_d, tell_d_up ]

module System is Master || P_1 || N_A || N_B || A_1 || B_1 || C || D

/* ---------------------------------------------------------------------- */

predicate NoResult is (result = DontKnow)
predicate NoNegativeResult  is (  ( (result = DontKnow) & (result' = DontKnow))
                                 |( (result = Yes)      & (result' = Yes) ))
predicate ResultsInDontKnow is ~( (result = DontKnow) & (result = Yes) )

judgment falseJudge is System |= NoResult
judgment badJudge   is System |= NoNegativeResult
judgment trueJudge  is System |= ResultsInDontKnow
```

Recent BRICS Report Series Publications