Heuristics for Hierarchical Partitioning with Application to Model Checking[⋆]

M. Oliver Möller[1] and Rajeev Alur[2]

[1] BRICS[⋆⋆]
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Århus C, Denmark
Fax.: (+45) 8942 3255
omoeller@brics.dk

[2] **University of Pennsylvania**
Computer & Information Science
Moore building
200 South 33rd Street
Philadelphia, PA 19104, USA
Fax.: (+1) 215 898-0587
alur@cis.upenn.edu

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 *NP*-complete. We present a greedy polynomial-time algorithm that approximates good solutions incrementally by local evaluation of a heuristic function. We argue for a heuristic function 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* [2]. The latter is applicable in reachability analysis and is included in a recent version of the Mocha model checking tool. We demonstrate performance and benefits of our method and use an asynchronous parity computer and an opinion poll protocol as case studies.

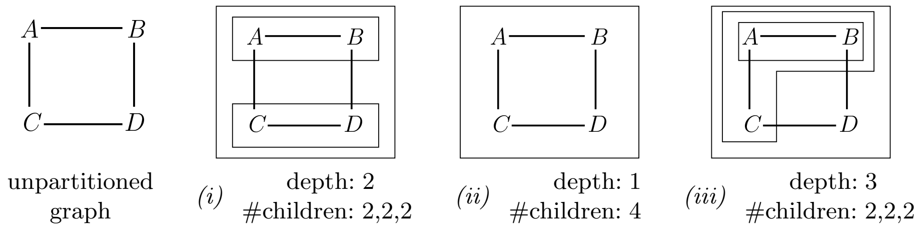
1 Introduction

Imposing a hierarchical structure on a collection of components is helpful in many contexts for different reasons, such as better understanding and better analysis.

Consider four items, call them A, B, C, and D. They may be connected in some way, say by a mutual dependency. Let us assume that this gives rise to a ring structure, like in Figure 1. Then, instead of viewing the system as a *set*

[⋆] A longer version of this paper is available as technical report [12].

[⋆⋆] Basic Research in Computer Science, Center of the Danish National Research Foundation.



unpartitioned graph

(i) depth: 2 #children: 2,2,2

(ii) depth: 1 #children: 4

(iii) depth: 3 #children: 2,2,2

Fig. 1. Different ways to hierarchically partition a square.

with 4 elements, we can understand it structured as $\{\{A, B\}, \{C, D\}\}$, like in *(i)*. Here, only two connections A-C and B-D need to be visible (or understood) at the top level. With the new compounds $\{A, B\}$ and $\{C, D\}$ we found a more abstract description of the same data, that can be refined on demand. Another possible partition is $\{\{A,\ D\},\ \{B,\ C\}\}$, but this requires all connections to be visible at the top-level, and thus should be rejected in favor of the first. We can partition in a hierarchic fashion: for example, $\{\{\{A,\ B\},\ C\},\ D\}$. In general, given a set, we partition it, and apply the process recursively to each set in the partition.

The set of distinguishable *hierarchical partitions* is adequately described as the set of rooted trees over leaf nodes $\{A,\ B,\ C,\ D\}$. In Figure 1, we draw these trees as cascading boxes that may contain other boxes. Every box corresponds to an intermediate node and the outermost box to the root. As a rule, we favor trees that have a low degree of branching and are nevertheless shallow. The diagrams *(ii)* and *(iii)* depict both not very good trees, since they are either too broad or too deep. Moreover, it is desirable to minimize dependencies among remote tree parts, i.e., the number of links crossing box boundaries should be low.

We can regard this as a general design problem, where trees form an architectural hierarchy over atomic units. This *modular* description helps to see the same system on different levels of abstraction or detail. The emphasis on modularity and hierarchy is a central theme in software engineering, particularly in software design notations such as Statecharts [10] and UML [3]. While the most appropriate hierarchical structure can best be chosen by the designer, automatically constructing a hierarchical partition is required if no manually chosen structure is available, or if the original structure is lost during translations between models (e.g., during the process of abstraction).

Formal verification is a field where structure is particularly useful, since it is generally considered infeasible to deal with unorganized descriptions. Structure helps to spot design flaws, but it can also be exploited to make algorithmic treatment more efficient, or even possible at all. Well-known examples for this are model checking problems. *Model checking* [4, 11] is a powerful technique for discovering inconsistencies in 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 once more the example with the four components. Let us interpret each atom as a process and each connection as the ability to synchronize on some action. We view the system hierarchically decomposed as in Figure 1 *(i)*. Tools such as the concurrency workbench [5] can analyze it in the following way. First take the product of processes A and B. Now their synchronization can be viewed as internal to this composite process, and we can apply a reduction based on weak bisimulation minimizing the size. Analogously, compose C with D and minimize. The obtained description is still adequate, since it shows *the same behavior* (modulo the internal synchronization), but questions about this behavior are algorithmically easier to answer.

An alternative method that benefits from a hierarchical structure is implemented in a recent version of the model checking tool Mocha, and will form the basis of the experiments in this paper. The technique, called *"Next" heuristic*, is a heuristic for on-the-fly search based on compressing unobservable transitions to a single meta-transition [2]. The basic idea is to describe the implementation P in a hierarchical manner, so that P is a tree whose leaves are atomic processes, and internal nodes compose their children and hide as many variables as possible. The basic reduction strategy, proposed by many researchers, is simple: while computing the successors of a state of a process, apply the transition relation repeatedly until a shared variable is accessed. This is applicable since changes to a private state are treated as stuttering steps. The benefit is greatly amplified by applying the reduction in a *recursive* manner exploiting the hierarchical structure, and has been shown to give significant reductions in space and time requirements, particularly for well-structured systems such as rings and trees.

As it turns out, the gain depends heavily on the hierarchical partition we impose. Ultimately, the run-time of the model checking algorithm can be seen as a measure on how to *compare* different choices. However, in practice we would look for qualifications that are faster to evaluate. And academically, run-time comparisons are too dependent on low-level implementation details to give clear analytic data. Thus we strive for a more abstract notion of comparison by means of a cost measure. For typical measures, the problem of finding the *best* hierarchical decomposition is likely to be *NP*-hard, and hence we must look for heuristics that are to be validated by experimentation.

In this paper, we strive to "discover" good hierarchies. We give a cost measure that allows us to compare hierarchical partitions, whenever the means of connection can be adequately described by a hypergraph. Determining the best structure for this measure is *NP*-complete. We present a greedy polynomial-time algorithm, that approximates good hierarchical partitions by local evaluation of a heuristic function. We corroborate applicability and usefulness via two case studies with our implementation of this algorithm in the Mocha verification tool. When applied to a tree-shaped topology, this results in significant time- and memory-savings. In the second class of examples, the run-time performance is not drastically improved.

Plan. The next section gives a formal definition of the problem and classifies its complexity. In section 3 we develop the algorithm to approximate good solutions.

n	1	2	3	4	5	6	7	8	9	10
$T(n)$	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 distinguishable tree-indexings.

Section 4 reports on experiments with sample problems. Section 5 reflects on the limitation of our method, contrasts it to related work and lists open problems.

2 The Tree-Indexing Problem

In the following, we describe systems as hypergraphs, where the atomic units correspond to vertices and their connections are represented by hyperedges. E.g., in a reactive module description every hyperedge would correspond to a variable shared by the modules it connects to. Hierarchical partitions introduce an additional tree structure on top of this hypergraph and are augmented with a cost value. We briefly treat combinatorial and computational complexity of finding a tree of minimal cost.

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. We draw hyperedges as branching lines. This coincides with common graph representation for the special case that every hyperedge is of size 2.

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. We draw internal nodes as polygons, all contained polygons and vertices $v \in \mathcal{C}$ are children of this node. The outermost polygon corresponds to the root. Every tree-indexing is qualified by a cost value dependent on $\mathcal{E}$. For instance, in Figure 1, the tree-indexing $\{\{A, B\}, \{C, D\}\}$ is better than $\{A, B, C, D\}$, and thus should be of lower cost.

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 [13]. 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. We can construct an equation, that $\hat{\varphi}(z)$ has to satisfy according to the theory of admissible constructions [7]. Every tree-indexing is either atomic, i.e. represented as z, or a set of at least two other tree-indexings, namely its children. This can be expressed using the admissible constructions $\mathfrak{Union}$ and $\mathfrak{Set}$.

$$\hat{\varphi}(z) \;=\; \mathfrak{Union}\,(z,\; \mathfrak{Set}\,(\hat{\varphi}(z),\, \text{cardinality} \geq 2)) \tag{1}$$

Equation (1) can be transcribed as follows.

$$\begin{array}{rrcl} & \hat{\varphi}(z) & = & z \;+\; \sum\limits_{k\geq 2} \frac{1}{k!} \cdot (\hat{\varphi}(z))^k \\ \iff & \hat{\varphi}(z) & = & z \;+\; \exp(\hat{\varphi}(z)) \;-\; \hat{\varphi}(z) \;-\; 1 \\ \iff & \exp(\hat{\varphi}(z)) & = & 2\,\hat{\varphi}(z) - z + 1 \end{array} \tag{2}$$

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 $T(n)$, as done in [6]. 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 a tree $\mathcal{T}$. 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

$$\mathit{depth\_cost}(\mathcal{T}) \;:=\; \begin{cases} 2 & \text{if } \mathit{depth}(\mathcal{T}) = 1 \\ \mathit{depth}(\mathcal{T}) & \text{otherwise}\,. \end{cases} \tag{3}$$

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

$$\mathit{cost}(\mathcal{T}) \;:=\; \sum_{e\in\mathcal{E}} \mathit{depth\_cost}(\mathcal{T}_e) \,\cdot\, |\mathit{leaves}\,(\mathcal{T}_e)\,| \tag{4}$$

For example, the tree-indexing *(i)* in Figure 1 has cost $2 \cdot 2 \cdot 2 + 2 \cdot 2 \cdot 4 = 24$, which is preferable to tree-indexings *(ii)* and *(iii)* with costs $4 \cdot 2 \cdot 4 = 32$ 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. Containment in *NP* holds, since we can guess any tree-indexing and compute its cost in (non-deterministic) polynomial time. A *NP*-hardness proof by reduction from MINIMUM CUT INTO EQUAL-SIZED SUBSETS is given in [12]. We expect EDGE-GUIDED TREE-INDEXING to remain *NP*-hard for other non-trivial definitions of a cost function like $\sum_e |\mathit{leaves}(\mathcal{T}_e)|$, though we do not have a proof for this. This precludes the possibility to determine an *optimal* tree-indexing in polynomial time[1] and suggests the application of heuristics in order to find a *reasonably good* tree-indexing efficiently.

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

```
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($Q$)                                   /* pick best candidate */
       let $\mathcal{A}'$ := fresh root node with children $t \in \mathcal{A}$
       $\mathcal{F} := (\mathcal{F} \setminus \mathcal{A}) \cup \{\mathcal{A}'\}$
       $\mathcal{E} := \mathcal{E} \setminus \{e \mid e \subseteq leaves(\mathcal{A}')\}$     /* 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 $\mathcal{F}$
```

Fig. 2. Incremental algorithm for constructing a tree-indexing.

3 A Greedy Algorithm to Partition Hierarchically

In this section we develop a greedy-style algorithm that constructs a tree-indexing by successively grouping together sets with strong correspondence. The choice of these candidates relies on heuristics, which make use of a key observation: strong correspondences are likely to represented by a large number of connections.

A schematic description of our proposed algorithm is given in Figure 2. The variable $\mathcal{F}$ is used to maintain a partial tree-indexing, i.e., a forest $\mathcal{F}$ with leaves $\mathcal{C}$. It is initialized as the forest with $|\mathcal{C}|$ trees, each consisting of a single node. The priority queue Q is ordered according to a rating function $\mathbf{r} : \wp^{\mathcal{D}} \times 2^{\mathcal{C}}_{\star} \rightarrow I\!R$. $\wp^{\mathcal{D}}$ is the set of forests over leaves $\mathcal{D} \subseteq \mathcal{C}$ and thus contains all possible sub-forests of $\mathcal{F}$. $2^{\mathcal{C}}_{\star}$ denotes a multi set of hyperedges and initially corresponds to $\mathcal{E}$. The top element of the queue is a subset of $\mathcal{F}$ with maximal $\mathbf{r}$-value.

The algorithm proceeds as follows. An initial set of candidates proposed for grouping together is inserted in the priority queue. Then a small number of executions of the while-loop follow. In each execution, the most promising candidate $\mathcal{A}$ is dequeued and the data is updated: in the forest, the trees in $\mathcal{A}$ are replaced by a tree with the fresh root $\mathcal{A}'$ and children $t \in \mathcal{A}$. Every set containing trees $t \in \mathcal{A}$ is removed from the priority queue and new candidates containing $\mathcal{A}'$ are inserted. Hyperedges $e \subseteq leaves(\mathcal{A}')$ are deleted, since they should not influence later selections.

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

Developing a good rating function. The local choice of the best candidate could be performed by means of the *cost* function defined in (4), i.e., by picking the candidate with lowest cost after clustering. We chose not to do so for two reasons. First, the specific cost function was derived such that an *NP*-complete problem could be encoded into it; for small variations of this definition, the proof failed - thus, this particular definition is somehow artificial. Second, we would like to tune the choice by means of parameters in the rating function. Doing this with the cost function would almost certainly destroy the provable *NP*-hardness, and thus the justification for the choice.

Instead, we develop a rating function subsequently by taking into account the—supposedly—crucial factors concerning the structure of the proposed candidate. 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 1 (cover number) *Let $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ be a hypergraph, $\mathcal{F}$ a forest over leaves $\mathcal{C}$, $\mathcal{A} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\} \subseteq \mathcal{F}$. The* cover number *of $\mathcal{A}$, in symbols $\langle\!\langle \mathcal{A} \rangle\!\rangle$, is defined as the number of hyperedges covered by the trees in $\mathcal{A}$.*

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

Though this value tells a lot about a candidate, it is isolated not a good guideline. Recall that the set $\mathcal{C}$ has naturally always the highest possible cover number $|\{\ell_e \,|\, e \in \mathcal{E}\}|$.

We relate the cover number to the *size* n of a candidate, where size matters in terms of *possible* connections, which is $\binom{n}{2} = \mathcal{O}(n^2)$. We propose the following rating function.

$$\mathbf{r}_{pref}(\mathcal{A}) \;:=\; \frac{\langle\!\langle \mathcal{A} \rangle\!\rangle}{|\mathcal{A}|^2} \tag{5}$$

In the following we refine $\mathbf{r}_{pref}$ by adding more structural information.

Def 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 $\mathcal{H}$.*

$$\mathit{touch}(\mathcal{A}) \;:=\; \left\{ \ell_e \,\middle|\, e \cap \mathit{leaves}(\mathcal{A}) \neq \emptyset \;\wedge\; e \not\subseteq \mathit{leaves}(\mathcal{A}) \right\}$$

Def 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*

$$\mathit{depth}\left(\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}\right) \;:=\; 1 + \max_{1 \leq i \leq k} \mathit{depth}(\mathcal{T}_i)$$

We do not want to cut out subsystems, that are multiply connected to the rest, i.e., those who share many hyperedges with their complement. This is reflected by the number of labels in the touch: if it is small, the candidate is more attractive. Also, it is perceivable that preference should be given to candidates with small depth. Hence we propose the following improved rating function.

$$\mathbf{r}^{+}_{pref}(\mathcal{A}) := \frac{\langle\!\langle \mathcal{A} \rangle\!\rangle}{|\mathcal{A}|^2} \;+\; \frac{\varepsilon_1}{|touch(\mathcal{A})|} \;+\; \frac{\varepsilon_2}{depth(\mathcal{A})} \tag{6}$$

The parameters ε_1 and ε_2 are supposed to be chosen small and positive. For the experiments in section 4.1, 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 (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 $touch(\mathcal{T}_1) \cap touch(\mathcal{T}_2) \neq \emptyset$.*

Clearly, 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 numbers. First computing interesting pairs and then extending them to candidates is an advantage.

The number of candidates can still be excessive. Consider a hyperedge connecting all vertices, then all pairs are interesting pairs. 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.

In the algorithm, the number of forests—initially n—decreases by one with each execution of the while loop. Operations like evaluating the rating function, testing $\mathcal{B} \cap \mathcal{A} \neq \emptyset$, and constructing new candidates containing $\mathcal{A}'$ can be assumed to be $\mathcal{O}(n)$, thus one execution of the while loop has the run-time bound $\mathcal{O}(n \cdot n^{k+1})$. The whole algorithm *partition_incrementally* has n executions of the while loop, which yields the polynomial bound $\mathcal{O}(n^{k+3})$ on its run-time.

4 Experimental Results

We implemented the algorithm from section 3 in an experimental version of the Mocha verification tool [1]. For symbolic (BDD-based) model checking, the

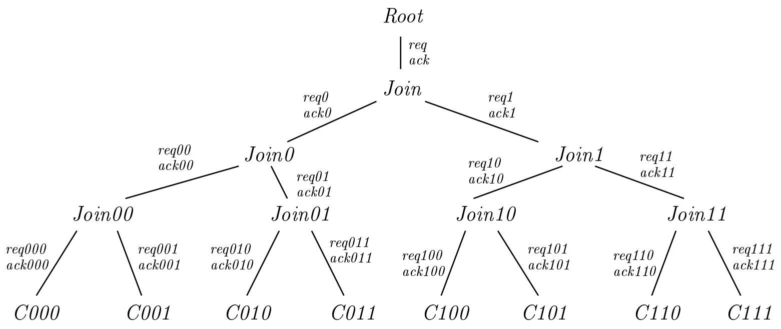


Fig. 3. Layout of an asynchronous parity computer with eight clients.

Java implementation makes use of native libraries. However, our experiments do not make use of this option and perform the check in a purely enumerative manner. Therefore, 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. A contingent of 128 MB of memory was allocated, run-times are in milliseconds. Together with an optimization in the enumerative check called *"Next" heuristic* [2], we are able to corroborate effectiveness and usability of our algorithm in some simple examples. We consider an asynchronous parity computer and an opinion poll protocol. The Mocha specifications are given in [12].

Note that in these experiments the checked property influences the obtained structure. In Mocha every property relies on *variables* of the system. These variables can not be hidden and therefore are neglected in the partition algorithm, i.e., they are ignored in the evaluation of the rating function.

4.1 Asynchronous Parity Computer

This example models a parity computer, designed as a binary tree (Figure 3). The leaf nodes are *Client* modules (abbreviated with C), communicating a binary value to the next higher *Join*. A simple hand-shake protocol is devised by the two variables *req* and *ack*. 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. When a client receives an acknowledgment, it is able to devise a fresh value.

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.

Reachability involves computing the successors of every state encountered, starting with the initial states. Consider the set S of all the processes. Then, successors of a state are computed by executing one step of one of the processes

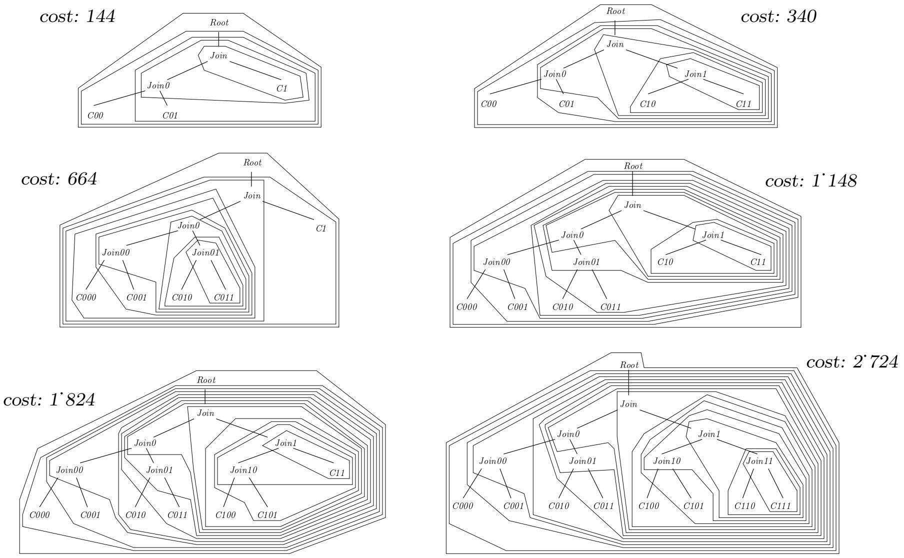


Fig. 4. Parity computers $N = 3, \ldots, 8$, partitioned with rating function $\mathbf{r}_{pref}$.

in S. Now suppose, we cluster the processes *Join00*, *C000*, and *C001* into one composite process called P, and replace these three processes in S with P. It is clear that the communication between *J00* with its children clients can be hidden from the rest of the system. Consequently, in reachability analysis of S, when we compute the successors due to execution of P, we can let the subprocesses in P repeatedly execute until *Join00* communicates with its parent *Join0*. This is formalized in MOCHA by substituting P by a construct *next Θ for P*, where atomic transitions correspond to sequences of transitions of P until a variable shared with the remaining system is accessed. The modified search yields an improved performance as it cuts down on unnecessary interleavings.[2] This scheme can be applied repeatedly. It should be clear that the effectiveness of the scheme depends on the hierarchical partition.

An intuitively good choice for this hierarchical partition is grouping together bottom up. Detecting this algorithmically is subtle. E.g., the difference between {*Join0*, *Join00*} and {*Client000*, *Join00*} is only minor, since both pairs cover exactly two variables. An incautious technique easily runs into errands, as to be seen in Figure 4. Using $\mathbf{r}_{pref}$ as rating function in the algorithm *partition_incrementally* leads to uncomfortably deep hierarchies.

The more sophisticated rating function $\mathbf{r}^{+}_{pref}$ performs far better, as seen in Figure 5. The parameters ε_1 and ε_2 were calibrated to $\varepsilon_1 := 1/1000$ and $\varepsilon_2 := 1/100000$, giving shallow structures a smaller bonus than those touching only few variables.

[2] A well-known method for reducing state-space in asynchronous systems is based on partial-order reductions [9]. The "Next" heuristic is incomparable to this method, see [2].

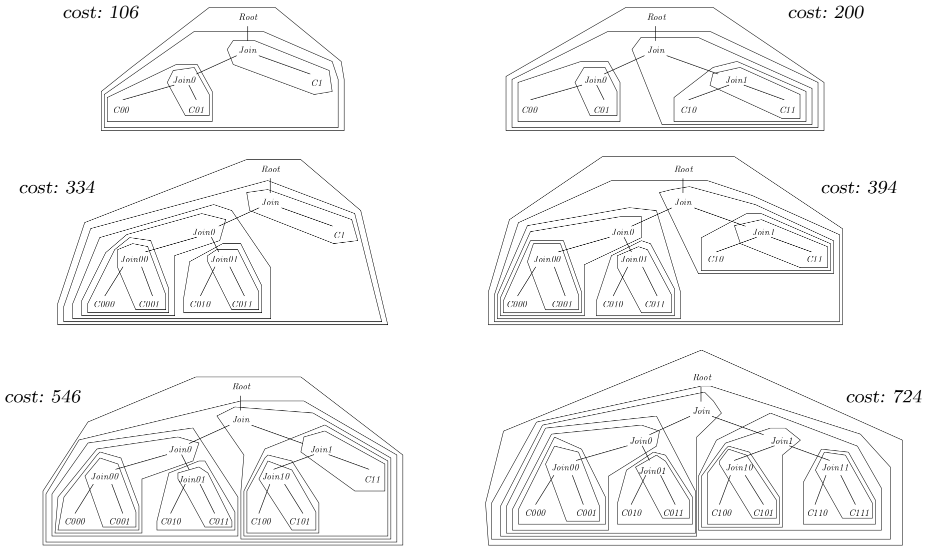


Fig. 5. $\mathbf{r}^{+}_{pref}$ yields shallower hierarchical partitions with lower cost values than $\mathbf{r}_{pref}$.

N	partition	|table|	check
3	162	95	1˙121
4	853	645	4˙921
5	740	1˙943	17˙086
6	2˙811	16˙045	161˙394
7	9˙928	58˙351	694˙834
8	47˙239	410˙901	5˙442˙315

Using $\mathbf{r}_{pref}$ as rating function

N	partition	|table|	check
3	105	51	973
4	148	117	1˙707
5	627	139	1˙780
6	2˙097	205	3˙000
7	10˙592	271	4˙395
8	50˙664	469	8˙322

$\mathbf{r}^{+}_{pref}$, $\varepsilon_1 := 10^{-3}$,$\varepsilon_2 := 10^{-5}$

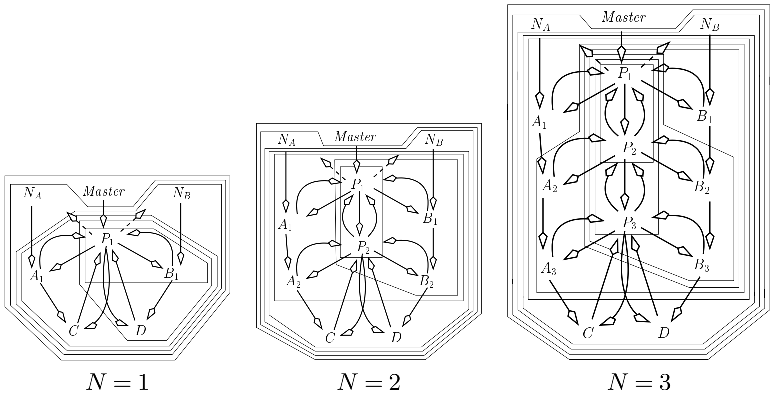
N	partition	|table|	check
3	57	51	404
4	75	117	1˙097
5	127	139	1˙726
6	516	205	2˙929
7	247	271	4˙364
8	342	469	8˙184

$\mathbf{r}^{+}_{pref}$ with $|\mathcal{A}| \leq 2$

Table 2. Parity computer: Comparison of two heuristic functions.

The deep hierarchical structure obtained by using $\mathbf{r}_{pref}$ lead to excessive number of explored states, whereas with $\mathbf{r}^{+}_{pref}$ the growth of the explored state space with increasing N is only moderate. This gap is also reflected by the significantly higher cost values. Table 2 shows the run-time data in detail. With "partition" we denote the preprocessing time used by *partition_incrementally* and "check" corresponds to the run-time of the model checking algorithm. The number of explored states is recorded under "|table|", Mocha keeps the states in a hash table. Note that the property we check does not hold, thus the model checking algorithm is able to abort without exploring all reachable states.

In the left and middle table, the time consumed for computing the hierarchical partition exceeds the model checking time for bigger examples. This is because we chose not to restrict the candidate size here, which yields a number of candidates increasing exponentially in N. The obtained hierarchical partitions



$N = 1$ $N = 2$ $N = 3$

Fig. 6. Opinion poll: Hierarchically partitioned according to rating function $\mathbf{r}^{+}_{pref}$.

in the middle and the right table are identical—due to the tree structure, the best rated candidate here is always of size two.

4.2 Opinion Poll Protocol

The second sample problem is meant to demonstrate the behavior of our heuristic in a setting, where there is no obviously preferable choice.

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 one other citizen and the poller who interviews them. For instance, A_{i+1} is influenced by A_i's and P_i's opinions, and A_1 is influenced by a random source N_A and P_1. The communication pattern is indicated by arrows in Figure 6.

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 without further preference, i.e., use $sig(\langle\!\langle\rangle\!\rangle)$ as rating function and consider only candidates of size 2, *b. pref*: Partition incrementally according to rating function $\mathbf{r}_{pref}$, and *c. pref+*: Partition incrementally according to rating function $\mathbf{r}^{+}_{pref}$. For the latter, we included the results of the preprocessing in Figure 6. It is interesting to note that sometimes triples were preferred to pairs.

The quantitative comparison is listed in Table 3. For the false properties *(i)* and *(ii)* the enumerative check is up to five times *slower*, if sophisticated heuristics are applied. Apparently it is more tedious to reach a counter-example

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

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

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

Table 3. Opinion poll protocol: Run-time comparison of three rating functions.

scenario here, if more structure is given. For the true property *(iii)*, the enumerative check speeds up by a third, when the "Next" heuristic is applied. For larger N the more sophisticated clustering techniques *pref* and *pref+* perform slightly better than *2-merge*.

5 Conclusion

We developed a notion of hierarchical partitions and introduced a method to compare different structures by means of a cost function. This is applicable, whenever the relationship of entities can be adequately described via hyperedges. For the presented cost function, the problem of determining an optimal hierarchical partition is *NP*-complete [12].

We presented a scalable greedy method to compute approximately good hierarchical partitions based on a heuristic rating function. We argued that—in order to achieve a good result—this function should be based on four criteria: number of covered hyperedges, size, number of occurring hyperedges, and structural depth. This is corroborated by qualitative and quantitative data. We implemented our algorithm in an experimental version of the Mocha model checking tool and measured its performance on small and medium sized examples.

It should be noted that 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 our algorithm can get caught in local optima. Moreover, optimality in the sense of least cost does not necessarily imply minimal time- or space-consumption when running a model checking algorithm. In general, we cannot expect to express such subtle behavioral properties of a system via a simple function, i.e., a function that is fast to evaluate.

Though our case studies suggest that in the rating function both touch and depth of the candidates should be taken into account, it remains open, *how* this should be reflected. The values for the parameters ε_1 and ε_2 in (6) were chosen according to fit the parity computer example. It would be desirable to investigate the impact of parameter changes in general, but we seem to lack apt mathematical means to do so.

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 [14]. 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. [8]), 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.

Open Problems. We noted that finding an optimal solution with respect to our cost function is *NP*-hard, but this does not preclude the existence of a 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.

Future Work. Our proposed method is not limited to Mocha, but can be applied in other settings where connected entities have to be structured or re-structured. The parameters can be adjusted accordingly. An interesting mean to make use of the obtained partitions could be to construct property preserving abstractions based on this particular hierarchy.

We believe that other areas of research can benefit from a more analytic quantification of good structures. As an example we list refactoring [15], where structures of software systems are conservatively modified according to a set of schema transformations, design patterns, or hot spots.

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

1. Rajeev Alur, Luca de Alfaro, Radu Grosu, Thomas A. Henzinger, Minsu Kang, Rupak Majumdar, Freddy Y.C. Mang, Christoph Meyer-Kirsch, and Bow-Yaw Wang. Mocha: A model checking tool that exploits design structure. *Proceedings of 23rd International Conference on Software Engineering*, 2001. See `www.cis.upenn.edu/~mocha/`.
2. 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.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
4. Edmund M. Clarke, Jr. and E. Allen Emerson. Synthesis of synchronization skeletons from branching time temporal logic. *Lecture Notes Comp. Sci.*, 131:52–71, 1982.
5. Rance Cleaveland, Joachim Parrow, and Bernhard 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.
6. Philippe Flajolet. A problem in Statistical Classification Theory, 1997. `http://pauillac.inria.fr/algo/libraries/autocomb/schroeder-html/`.
7. 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.
8. 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.
9. Patrice Godefroid, Doron Peled, and Mark Staskauskas. Using partial order methods in the formal validation of industrial concurrent programs. *IEEE, Transactions on Software Engineerings*, 22:496–507, 1996.
10. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 1987.
11. Gerard 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.
12. M. Oliver Möller and Rajeev Alur. Heuristics for hierarchical partitioning with application to model checking. Research Series RS-00-21, BRICS, Department of Computer Science, University of Aarhus, August 2000. 30 pp, available online at `http://www.brics.dk/RS/00/21/`.
13. Ernst Schröder. Vier combinatorische probleme. *Zentralblatt. f. Math. Phys.*, 15:361–376, 1870.
14. Naveed Sherwani. *Algorithms for VLSI Physical Design Automation - 2nd Edition*. Kluwer Academic Publishers, Norwell, USA, 1995.
15. Lance Tokuda and Don Batory. Evolving Object-Oriented Designs with Refactorings. In *Proceedings of ASE-99: The 14th IEEE Conference on Automated Software Engineering*. IEEE CS Press, October 1999.