Finding Boundary Elements in Ordered Sets with Application to Safety and Requirements Analysis

Jaroslav Bendík(✉), Nikola Beneš, Jiří Barnat, and Ivana Černá

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbendik,xbenes3,barnat,cerna}@fi.muni.cz

Abstract. The motivation for this study comes from various sources such as parametric formal verification, requirements engineering, and safety analysis. In these areas, there are often situations in which we are given a set of configurations and a property of interest with the goal of computing all the configurations for which the property is valid. Checking the validity of each single configuration may be a costly process. We are thus interested in reducing the number of such validity queries. In this work, we assume that the configuration space is equipped with a partial ordering that is preserved by the property to be checked. In such a case, the set of all valid configurations can be effectively represented by the set of all maximum valid (or minimum invalid) configurations w.r.t. the ordering. We show an algorithm to compute such boundary elements. We explain how this general setting applies to consistency and redundancy checking of requirements and to finding minimum cut-sets for safety analysis. We further discuss various heuristics and evaluate their efficiency, measured primarily by the number of validity queries, on a preliminary set of experiments.

Keywords: Requirements analysis · Formal verification · Safety analysis

1 Introduction

The motivation of this work comes from various source areas, such as parametric formal verification, requirements engineering, safety analysis, or software product lines. In these areas, the following situation often arises: We are given, as an input, a set of configurations and a property of interest. The goal is to compute the set of all the configurations that satisfy the given property. We call such configurations valid. As a short example, one may imagine a system with tunable parameters that is to be verified for correctness. The set of configurations, in that case, is a set of all possible parameter values and the goal is to find all such values that ensure the correctness of the given system. If we are given a method to ascertain the validity of a single configuration, we could try running the method repeatedly for each configuration to obtain the desired result. In the

© Springer International Publishing Switzerland 2016
R. De Nicola and E. Kühn (Eds.): SEFM 2016, LNCS 9763, pp. 121–136, 2016.
DOI: 10.1007/978-3-319-41591-8_9

case of an infinite set of configurations, this approach does not terminate, and we get at most a partial answer. However, even if the configuration space is finite, checking configurations one by one may be too costly. We are thus interested in reducing the number of validity checks in the finite case.

Although such reduction might be impossible in general, we focus on problems whose configuration space is equipped with a certain structure that is preserved by the property of interest. This may then be exploited in order to check a smaller number of configurations and still obtain the full answer. The desired structure is a set of dependencies of the form: "If configuration A violates the property then configuration B does too." Mathematically, we can either view such structure as a directed acyclic graph of those dependencies, or as a partial ordering on the set of all configurations induced by this graph. Viewed as an ordered set, the set of all the valid configurations can be effectively represented by the set of all the maximal valid (alternatively, minimal invalid) configurations.

We are interested in finding this boundary between valid and invalid configurations while minimising the number of validity queries, i.e. the potentially costly checks whether a given configuration satisfies the property.

We are not aware of any previous work which deals with exactly the same problem as we do. The most related problems can be found among the Constraint Satisfaction Problems (CSPs) where a satisfiability of a set of constraints is examined. When a set of constraints C is infeasible the most common analysis is the maximum satisfiability problem (MaxSAT, MaxCSP), which asks for a satisfiable subset of C with the greatest possible cardinality. Our problem is different from MaxSAT and more related to the maximum satisfiable subset problem (MSS) that considers maximality in the ordering sense instead of maximum cardinality. The goal of MSS is to find a subset of C that is satisfiable, and that becomes unsatisfiable if any other constraint is added to this subset. Similarly, one can define the minimum unsatisfiable subset problem (MUS).

Both MSSes and MUSes describe the boundary between the satisfiable and unsatisfiable subsets of C and both these problems were recently addressed in works [1,3,6,15,16]. To solve the problem, the papers use different approaches like the duality that exists between MUSes and MSSes [1,16] or parallel enumeration from bottom and top [3]. In [15] authors unify and expand upon the earlier work, presenting a detailed explanation of the algorithm's operation in a framework that also enables clear comparisons. Paper [6] describes an MUS extractor tool MUSer2 which implements a number of MUS extraction algorithms.

Subsets of a set of requirements are naturally ordered by the subset relation, thus our approach can be also used to solve these problems. We deal with a more general problem as we consider arbitrary graphs instead of the hypercube graphs representing subsets of requirements. Our approach has thus a wider area of potential usage. Furthermore, as is explained in Sect. 4, in the case of hypercubes our approach can be competitive with the state-of-the-art tool Marco [15].

Safety Analysis. The safety analysis techniques are widely used during the design phase of safety-critical systems. Their aim is to assure that the systems provide prescribed levels of safety via exploring the dependencies between

a system-level failure and the failures of individual components. Traditionally, the various safety analyses are done manually and are based on an informal model of the systems. This leads to the process being very time-consuming and the results being highly subjective. The desire to alleviate such issues somewhat and to make the process more automated led to the development of Model-Based Safety Analysis (MBSA) approach [13]. This approach assumes the existence of a system model that is extended by an error model describing the way faults may happen and propagate throughout the system. One of the problems solved in MBSA is the computation of the so-called minimal cut-sets for a given failure, i.e. the minimal sets of low-level faults that cause the high-level failure to manifest in the system.

One can map the minimal cut-sets problem to our setting easily. The configurations are the possible sets of faults that may be enabled in the extended system model, their ordering is given by set inclusion. Note that there might be dependencies between some of the faults, which means that not all sets of faults are considered to be possible. The property of interest is the non-existence of failure and the valid configurations are exactly those sets of faults that do not cause the failure to happen. Clearly, in this case, the minimal cut-sets correspond exactly to the minimal invalid configurations. This means that the problem can be solved using our approach.

To illustrate the application on a simple example, we consider an avionics triplex sensor voter, described in [9]. The voter gains measurement data from three sensors as well as information whether the sensors are operational. It computes the differences between the sensor data and detects persistent miscompare, i.e. situations where two sensors differ above a certain threshold for a certain amount of time. If all three sensors are operational and two pairs of sensors have persistent miscompare, the common sensor is marked as invalid and data is no longer received from that sensor. If just two sensors are operational, a persistent miscompare between the two means that the output data is considered invalid.

For simplicity, let us assume that there are two kinds of faults per sensor and let us call these fault A and fault B. Fault A causes the sensor to transmit wrong data while fault B causes the sensor to stop working completely. Note that we may assume that both faults cannot occur on the same sensor, as once fault B happens, the occurrence of fault A is irrelevant. In general, we thus have six possible faults and 27 sets of faults to be checked, including the empty set of faults. However, as the situation of sensors is symmetrical, we may get rid of this symmetry and simply count the number of fault-A sensors and fault-B sensors instead. This situation is illustrated in Fig. 1. The nodes in the graph represent the various fault configurations: $\emptyset$ represents that no faults occur, AB represents that fault A occurred on one sensor and fault B occurred on another sensor, etc. The graph is created from the inclusion ordering on the fault situations.

Let us now consider the failure to deliver data to the output. As explained above, the voter fails to deliver output if either all sensors stopped working or have been eliminated, or if there are just two sensors working with persistent miscompare. We assume that the persistent miscompare situation is detected

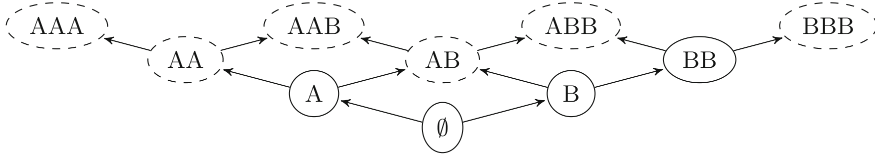

Fig. 1. Illustration of the safety analysis example

once at least one of a pair of sensors starts transmitting wrong data, i.e. fault A occurred on that sensor. For this reason, the minimum invalid configurations (i.e. the minimal cut-set) are AA, AB, and BBB, while the maximum valid configurations are A and BB.

Requirements Analysis. Establishing the requirements is an important stage in all development. Although traditionally, software requirements were given informally, recently there has been a growing interest in formalising these requirements [12]. Formal description in a kind of mathematical logic enables various model-based techniques, such as formal verification. Moreover, we also get the opportunity to check the requirements earlier, even before any system model is built. This so-called requirements sanity checking [3] aims to assure that a given set of requirements is consistent and that there are no redundancies. If inconsistencies or redundancies are found, it is usually desirable to present them to the user in a minimal fashion, exposing the core problems in the requirements. As redundancy checking can be usually reduced to inconsistency checking [2], the goal is thus to find all minimal inconsistent subsets of requirements. Such a problem may be clearly seen as an instance of our setting, where the configurations are sets of requirements and the ordering is given by the subset relation.

We illustrate the inconsistency checking on an example. Assume that we are given a set of four requirements. These requirements consider one particular component in a system and constrain the way the component is used. We formalise the requirements using the branching temporal logic CTL [8]. In the formulae we use the atomic propositions q denoting that a *query* has arrived, r denoting that the component is *running*, and m denoting that the system is taken down for *maintenance*. Our first requirement states that whenever a query arrives, the component has to become active eventually, formally $\varphi_1 := \mathbf{AG}(q \rightarrow \mathbf{AF}\, r)$. The second requirement states that once the component is started, it may never be stopped. This may be a reasonable requirement e.g. if the component's initialisation is expensive, formally $\varphi_2 := \mathbf{AG}(r \rightarrow \mathbf{AG}\, r)$. The third requirement states that the system has to be taken down for maintenance once in a while. This also means that the component has to become inactive at that time. This is formalised as $\varphi_3 := \mathbf{AG}\,\mathbf{AF}\,(m \wedge \neg r)$. Our last requirement states that after the maintenance, the system (including the component we are interested in) has to be restarted, formally $\varphi_4 := \mathbf{AG}(m \rightarrow \mathbf{AF}\,(\neg m \wedge r))$. The situation is illustrated in Fig. 2. We discover that there is one minimum inconsistent subset of the four requirements, namely $\{\varphi_2, \varphi_3, \varphi_4\}$, and that there are three maximum

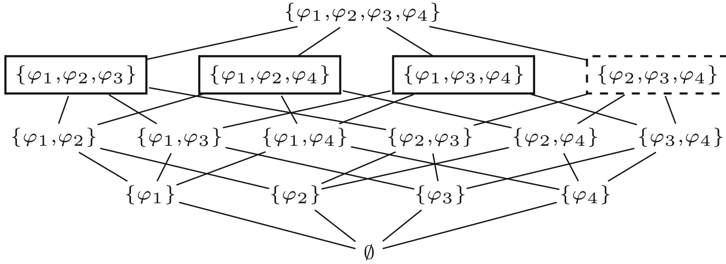

Fig. 2. Illustration of the requirements analysis example. The subset with dashed outline is the maximum inconsistent one, the subsets with solid outline are the maximum consistent ones.

consistent subsets of the requirements, namely $\{\varphi_1, \varphi_2, \varphi_3\}$, $\{\varphi_1, \varphi_2, \varphi_4\}$, $\{\varphi_1, \varphi_3, \varphi_4\}$. The consistency of the first set $\{\varphi_1, \varphi_2, \varphi_3\}$ might be surprising, as one would suspect the pair of requirements φ_2 and φ_3 to be the source of inconsistency. However, the first three requirements can hold at the same time – in systems where no queries arrive at all. In these situations we say that the requirements hold *vacuously*. There are ways of dealing with vacuity, such as employing the so-called vacuity witnesses [5].

Note that although in this example, the space of all sets of requirements had the particular shape of a hypercube, this might not always be the case. We might sometimes be interested in certain subsets of requirements instead of all of them. Such a situation may arise e.g. if there are some known implications between the requirements. Consider the example above with the added requirement that once the component is started, it may only stop after 1 h. This requirement is clearly implied by φ_2 and we would therefore omit all subsets that contain both φ_2 and this new requirement. Another way of obtaining a non-hypercube requirements graph is when considering requirements for several components at once in a component-based or software product line setting. In such cases, some of the components or product features may be incompatible and it thus only makes sense to consider subsets of requirements that reason about compatible components.

Outline of the Paper. The rest of this paper is organised as follows. In Sect. 2 we present the basic definitions and preliminaries and state our problem formally. In Sect. 3 we present our new algorithm to solve the problem and discuss several variants and heuristics of it, as well as we analyse its complexity. The algorithm is then evaluated on a set of experiments in Sect. 4 and the paper is concluded in Sect. 5.

2 Preliminaries and Problem Statement

In this section, we recall some basic notions that we use later in the paper. We also introduce the formalism of annotated directed acyclic graphs that forms the basic setting for our problem.

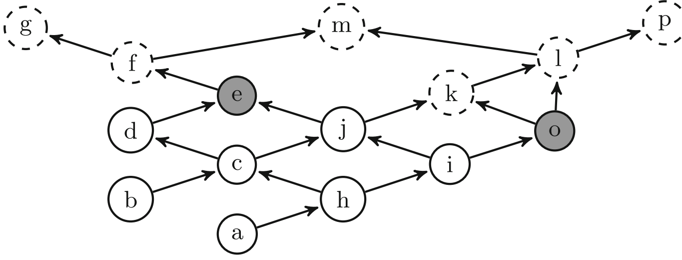

Fig. 3. An example of an ADAG, the dashed vertices are the invalid ones, the grey vertices are the maximum valid ones.

Definition 1 *(Directed Acyclic Graph). A* directed graph *G is a pair (V, E), where V is a finite set of vertices and $E \subseteq V \times V$ is a set of edges. An edge (u, v) is an* outgoing *edge of the vertex u and an* incoming *edge of the vertex v. The* indegree *(*outdegree*) of a vertex v is the number of incoming (outgoing) edges of v. A* path *from a vertex u to a vertex v in G is a sequence $\langle v_0, v_1, \cdots v_k \rangle$ of vertices such that $v_0 = u$, $v_k = v$, $k > 0$ and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \cdots, k-1$. We say that v is reachable from u if there is a path in G from u to v.*

A directed graph $G = (V, E)$ is called a directed acyclic graph (DAG) *if there is no path $\langle v_0, v_1, \cdots v_k \rangle$ in the graph such that $v_0 = v_k$. A DAG induces a strict partial order relation $\sqsubset_G$ on its vertices as follows: $u \sqsubset_G v$ if v is reachable from u. A vertex v is said to be a* minimum vertex *in G if there is no u such that $u \sqsubset_G v$. Dually, a vertex u is a* maximum vertex *in G if there is no v such that $u \sqsubset_G v$.*

Definition 2 *(Chain Cover). A* chain *in a DAG G with its induced relation $\sqsubset_G$ is a sequence of one or more vertices $\langle v_0, v_1, \ldots, v_k \rangle$ such that $v_0 \sqsubset_G v_1 \sqsubset_G \cdots \sqsubset_G v_k$. A* chain cover *of a DAG is a set of chains $C = \{c_1, \cdots, c_l\}$ such that each vertex is included in exactly one chain from C. A* minimum chain cover *is a chain cover containing the fewest possible number of chains. Note that the minimum chain cover is not given uniquely.*

Definition 3 *(Annotated DAG). An* annotated directed acyclic graph (ADAG) *is a pair $(G, valid)$, where $G = (V, E)$ is a directed acyclic graph and $valid : V \rightarrow Bool$ is a validation function. The validation function is* monotone *on V, which means that for every pair $u, v \in V$ if $u \sqsubset_G v$ and $valid(u) = false$ then $valid(v) = false$.*

The problem we are interested in can be stated as finding either a set of maximum valid vertices or a set of minimum invalid vertices. We present an algorithm to obtain the former. However, the algorithm can be also used to obtain the latter, as the two formulations are dual.

Definition 4 *(Maximum Valid Vertex and Cut). Let $\mathcal{G} = ((V, E), valid)$ be an ADAG. A vertex $u \in V$ is a* maximum valid vertex *of G iff $valid(u) = true$ and $\forall v \in V$ such that $u \sqsubset_G v$ is $valid(v) = false$.*

The maximum valid cut *of $\mathcal{G}$ is a set of all its maximum valid vertices.*

Problem Formulation. Given an ADAG $\mathcal{G} = ((V, E), valid)$, find the *maximum valid cut* of $\mathcal{G}$.

As mentioned in the introduction, evaluating the function *valid* on a single configuration (a single vertex of the ADAG) might be an expensive operation. Therefore, our aim is to propose an algorithm minimising the number of evaluations of the *valid* function even for the price of the increased complexity of the algorithm with respect to the number of operations over the graph.

The problem formulation assumes that the graph is acyclic and that the validation function is monotone. We might, however, be also interested in cases where one of these preconditions is violated. We postpone the discussion of these possibilities to Sect. 3.5.

3 Algorithm

A naive solution of the maximum valid cut problem for a given ADAG G would be to evaluate the *valid* function on each vertex, compute the $\sqsubset_G$ relation for valid vertices, and choose the maximum ones. In this naive approach, the *valid* function is called once per each vertex.

3.1 Chain-Based Algorithm

Instead of dealing with each vertex of G separately we build our solution on a decomposition of G into a set of chains and we use the fact that the validation function is monotone. The algorithm takes as an input an ADAG G and one of its chain covers C. Then it iteratively handles chains and removes those vertices which cannot be the maximum valid ones.

From the definition, each vertex of the maximum valid cut of G belongs to exactly one chain from C. Moreover, every chain contains at most one maximum valid vertex of the graph and this vertex is at the same time the maximum valid vertex of the chain. Let us note that the opposite implication does not hold generally, the maximum valid vertex of a chain may not be a maximum valid vertex of the whole graph. Therefore, the set of maximum valid vertices of individual chains contains the maximum valid cut as its subset.

Let $c = \langle v_0, v_1, \cdots, v_l \rangle$ be an arbitrary chain of C. To find the maximum valid vertex v_h of this chain we use binary search. We take the *middle* vertex c_{mid} of c, $c_{mid} = v_{\lceil \frac{l}{2} \rceil}$ and evaluate the *valid* function on c_{mid}. If c_{mid} is valid, then we know for sure that none of the lower vertices from c can be the maximum valid vertex of this chain. In the other case, we claim that none of the higher vertices from c can be maximum valid vertex. This allows us to reduce c into half and recursively repeat the procedure. We finish with a chain consisting of only one vertex v_i. If v_i is a valid vertex then it is the maximum valid vertex of c, otherwise c does not have any valid vertex at all.

Once we have applied the binary search on each chain from C, we have the set H of maximum vertices of these chains. To obtain the maximum valid cut of G from H we just compute the $\sqsubset_G$ relation for each pair from H and remove from H all those vertices that are not maximum w.r.t. $\sqsubset_G$.

For an illustration of the chain based algorithm, assume that we are given the graph from Fig. 3 and as a chain cover we take these chains: $\langle b, c, d, e, f, g\rangle$, $\langle a, h, j, k, m\rangle$, $\langle i, o, l, p\rangle$. The vertices e, j, o are found to be the maximum valid vertices of these chains and the $\sqsubset_G$ relation is computed for these three vertices. Vertex j is found to be lower than e and vertices o, e are mutually unreachable, hence $\{e, o\}$ is the resulting maximum valid cut.

The number of calls to *valid* in this algorithm depends on the number of chains in C and the number of calls used in the binary searches. The number of calls is logarithmic in the length of the chain in every binary search. Therefore, the total number of calls is $\mathcal{O}(|C| \log L)$ where $|C|$ is the number of chains in C and L is the length of the longest chain in C.

Note that there are algorithms such as [7,11] that compute the minimum chain cover of a given graph. We may thus make use of these algorithms to reduce the number of chains that need to be processed by this algorithm.

3.2 Cutoff-Based Algorithm

We now improve the efficiency of our algorithm by decreasing the chain lengths and possibly eliminating some of the chains completely. The main idea makes use of the fact that a vertex v_i is recognised as the maximum valid vertex of a chain $c = \langle v_0, v_1, \cdots, v_i, \cdots, v_l\rangle$ (if c has any). From this we can deduce that not only vertices from c lower than v_i cannot belong to the maximum valid cut, but neither do any vertices from G lower than v_i. Symmetrically, none of the vertices from G higher than v_{i+1} can belong to the maximum valid cut. Therefore, we can remove all vertices lower than v_i and higher than v_{i+1}, including v_{i+1}, from all chains and thus reduce their size and possibly the number of *valid* calls in the future.

Definition 5 *(Cutoff Transformation). Let G be an ADAG and C its chain cover. Let $c = \langle v_0, v_1, \cdots, v_i, \cdots, v_l\rangle$ be a chain from C and let v_i be its maximum valid vertex. Then the* cutoff *of G is a pair $\overline{G}$ and $\overline{C}$ generated from G and C, respectively, by removing:*

- *vertices which are lower than v_i,*
- *vertices which are higher than v_{i+1}, and*
- *the vertex v_{i+1}.*

In case that c does not have a maximum valid vertex we define the cutoff *of G to be a tuple $\overline{G}$ and $\overline{C}$ created from G and C, respectively, by removing:*

- *vertices which are higher than v_0, and*
- *the vertex v_0.*

As this vertex removal may make some chains empty, we also remove the empty chains from $\overline{C}$.

Theorem 1 *(Cutoff Property). Let G be an ADAG, C its chain cover, and $\overline{G}, \overline{C}$ be their cutoff. Then graphs G and $\overline{G}$ have the same maximum valid cuts, $\overline{C}$ is a chain cover of $\overline{G}$, and $|C| \geq |\overline{C}|$.*

```
MaxValid(c = ⟨v_0, v_1, ..., v_l⟩, IsValid())
 1  if c is empty
 2     then return nil
 3  middle ← ⌈l/2⌉
 4  if IsValid(v_middle)
 5     then x ← MaxValid(⟨v_middle+1, ..., v_l⟩, IsValid())
 6          if x = nil
 7             then return middle
 8             else return x
 9     else
10          return MaxValid(⟨v_0, ..., v_middle−1⟩, IsValid())
```

```
Cutoff(G = (V, E), c = ⟨v_0, v_1, ..., v_l⟩, i)
1  if i ≠ nil
2     then set v.cand = false for each v ∈ V lower than v_i
3          set v.cand = false for each v ∈ V higher than v_i+1
4          set v_i+1.cand = false
5     else set v.cand = false for each v ∈ V higher than v_0
6          set v_0.cand = false
```

```
MaxValidCut(G = (V, E), IsValid())
1  set v.cand = true for each v ∈ V
2  compute the relation ⊏_G
3  ChainCover ← MinimumChainCover(G)
4  for each chain ∈ ChainCover
5       do ProcessChain(G, IsValid(), chain)
6  return V
```

```
ProcessChain(G, IsValid(), c)
1  remove from c all vertices v with v.cand = false
2  index ← MaxValid(c, IsValid())
3  Cutoff(G, c, index)
```

Algorithm 1. Maximum Valid Cut Algorithm

Theorem 2 *(Maximal Cut Property). Let G be an ADAG and C its chain cover. Let us apply step by step the cutoff transformation on all chains from C and let $\overline{G}$ and $\overline{C}$ be the resulting graph and its chain cover respectively. Then every chain in $\overline{C}$ is just a single vertex and $\overline{C}$ is exactly the maximum valid cut of G.*

The algorithm based on the cutoff transformation is shown as Algorithm 1. The algorithm assumes that the reachability relation $\sqsubset_G$ is pre-computed. The relation is used both for computing the minimum chain cover and when detecting lower and higher vertices, however, bread-first-search can be also used for this detection. Instead of removing vertices from the graph we just mark them with a binary flag *cand* (for candidate) initially set to *true*. Once we have discovered that a vertex cannot be a maximum valid one, the flag is set to *false*.

Contrary to the previous algorithm based on chains, once the algorithm based on cutoffs processes the last chain from the chain cover of the original graph G,

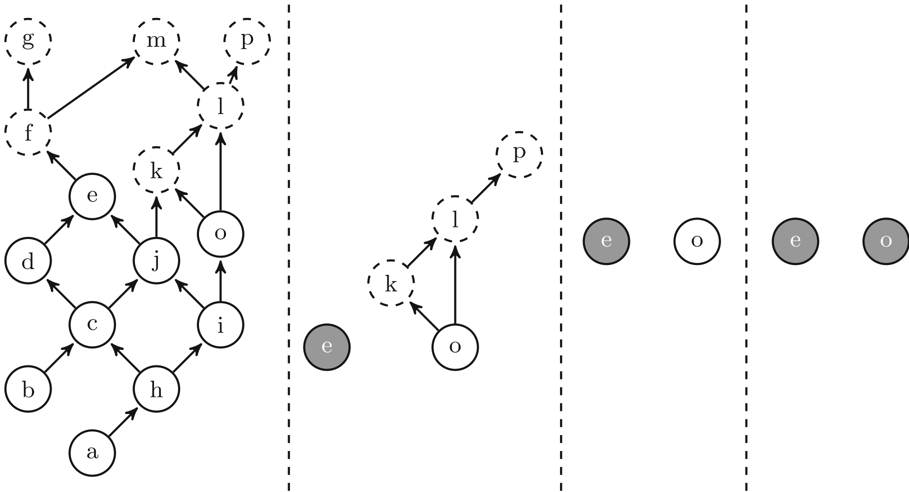

Fig. 4. Illustration of the cutoff based algorithm. The graph is covered with three chains $\langle b, c, d, e, f, g\rangle$, $\langle a, h, j, k, m\rangle$, $\langle i, o, l, p\rangle$ and they are processed in this order. At first, the vertex e is found to be the maximum valid vertex of the first chain and the consequently made cutoff reduces the set of chains to $\langle e\rangle$, $\langle k\rangle$, $\langle o, l, p\rangle$. In the next step, the chain $\langle k\rangle$ is processed and no valid vertex on this chain is found, but the cutoff is made and the set of chains is reduced to $\langle e\rangle$, $\langle o\rangle$. In the last step, the chain $\langle o\rangle$ is processed and o is found to be valid. The result of the third cutoff is the maximal valid cut $\{e, o\}$. The grey nodes are nodes which have already been determined to be valid ones.

the set C contains exactly the maximum valid cut of G and no other computation is needed.

Figure 4 illustrates the cutoff based algorithm on the graph from Fig. 3. The cutoffs significantly reduce the space of vertices that can be maximum valid ones. After processing of the first two chains only two vertices are left as the possible maximum valid ones.

3.3 Complexity

The time complexity analysis of the cutoff algorithm is given w.r.t. the size of the graph $G = (V, E)$ and we separately evaluate the number of *valid* calls and the number of all other operations.

The number of calls to the *valid* function depends on the number of chains in C and the number of calls used in the binary searches. The total number of calls is in the worst case the same as with the algorithm based on chains, i.e. $\mathcal{O}(|C| \log L)$ where $|C|$ is the number of chains in C and L is the length of the longest chain in C. Note that the size of the *minimum* chain cover can be bounded due to Dilworth's theorem.

Theorem 3 *(Dilworth's Theorem* [10]*). The size of the minimum chain cover of graph G equals to the size of a maximum number of pairwise unrelated elements, where u is unrelated to v if neither $u \sqsubset_G v$ not $v \sqsubset_G u$.*

To evaluate the overall complexity of the algorithm we denote by T_{valid} the time needed for one evaluation of *valid*.

The reachability relation $\sqsubset_G$ is in fact equal to the transitive closure of the graph and can be computed in $\mathcal{O}(|V| \cdot |E|)$ with the help of, e.g., depth-first search starting from each node of the graph.

The procedure ProcessChain first removes from the chain all vertices that have been recognised as not maximum valid in some of the previous cutoff transformations. When starting the MaxValidCut algorithm, each vertex is included in exactly one chain of the chain cover. Each vertex is removed at most once, hence the overall number of removals is bounded by the size of V and the complexity of the removals only is $\mathcal{O}(|V|)$.

The procedure MaxValid is an analogy of the binary search. It calls the validation function on the middle vertex of the given chain c, splits the chain into two halves, and recursively continues on one of these halves. The complexity of MaxValid is $\mathcal{O}(T_{valid} \cdot \log |c|)$ where $|c|$ is the length of c. The procedure is called once for each chain of the chain cover C of G resulting in the overall complexity of $\mathcal{O}(T_{valid} \cdot |C| \cdot \log L)$ where L is the length of the longest chain from C.

The procedure Cutoff marks those vertices which cannot be maximum valid ones. Either bread-first-search or the $\sqsubset_G$ relation can be used to detect the vertices, which should be marked, and each vertex is marked as *false* at most once. Therefore all the markings (including the initialisation) take time $\mathcal{O}(|V|)$.

The most time consuming part of the algorithm (excluding the *valid* calls) is the computation of the minimum chain cover taking time $\mathcal{O}(|C| \cdot |V|^2)$. For details and complexity analysis please refer to [7, 11]. The total time complexity of the cutoff algorithm is thus $\mathcal{O}(|V|^3 + T_{valid} \cdot |C| \cdot \log L)$.

3.4 Heuristics

The cutoff algorithm works with the minimum chain cover, however, the algorithm does not prescribe the order in which individual chains are processed. Each cutoff transformation affects the chains that have not been processed yet. Therefore the order in which the chains are processed affects the total number of calls to the validation function.

Cutting Power Based Heuristics. The order which minimises the number of calls to the validation function cannot be determined without the information which vertices are valid and which are not. Instead, for each chain c we can identify the minimum and the maximum number of vertices that can be cut off as a result of its processing. Let us define for each vertex v_i from the chain $\langle v_0, v_1, \ldots, v_l \rangle$ its *cutting power* as the number of vertices of G lower than v_i plus the number of vertices higher than v_{i+1} plus 1 (for vertex v_{i+1}). Then the *maximum cutting power of chain c* is the maximum of cutting powers of its vertices.

Average and median cutting power of a chain can be defined in a similar way. Cutting powers of vertices can be used to propose several heuristics decreasing the number of calls to the validation function.

The first heuristic sorts the chains in descending order according to their maximum cutting powers. This heuristic can lead to a large reduction of the graph while processing the first few chains. However, this happens only if the vertices with maximum cutting power are the maximum valid vertices of these chains.

As the second heuristic we propose to compute for each chain c its average cutting power which equals to the arithmetic mean of the cutting powers of its vertices. The heuristic sorts the chains in descending order according to their average cutting power. A similar heuristic is to order the chains according to the median of the cutting powers of its vertices. These two heuristics can speed up the average performance of the algorithm.

Note that to compute the cutting power of a vertex we need to know the reachability relation of the graph. The reachability relation is pre-computed when the minimum chain cover is constructed. The only additional computation required by the heuristics is thus the sorting which takes $\mathcal{O}(|C| \cdot \log |C|)$ time and does not increase the asymptotic complexity of the cutoff algorithm.

All heuristics can be improved if we recompute the cutting powers of vertices and sort the chains after each cutoff transformation. However, this requires recomputation of the reachability relation which is rather expensive and increases the complexity of the algorithm. As explained in the introduction, our goal is to minimise the number of calls to the validation function as it is assumed to be a very expensive operation. When choosing the appropriate heuristic we have to trade off between the number of validation function calls and the complexity of the heuristic.

Cutting Power Approximation. Yet another possibility is to approximate the cutting power of vertices by some easily computable characteristic. For instance, we can take the outdegree of a vertex as a high outdegree can indicate high cutting power. The same holds for the indegree of a vertex. Again, we can sort chains according to out/indegrees, average degree or median. On the one hand, this approach could be less effective than the approaches based on cutting powers. On the other hand, it is relatively cheap and affords to recompute the ordering after each cutoff transformation.

Online Computed Chains. As the precomputation of the minimum chain cover is rather expensive, our last heuristic drops this precomputation. The chains are instead computed on the fly. To construct a chain we take an arbitrary unprocessed vertex (i.e. a vertex whose validity is not known yet) and by following its unprocessed predecessors and successors we extend it to a chain. This chain is then processed as described in the cutoff algorithm and we repeat this process as long as there are some unprocessed vertices. We call this heuristic the *online heuristic*. Obviously, the disadvantage of this approach is that the number of the on-the-fly constructed chains can be much higher than the size of the minimal chain cover. However, if we precompute the minimal chain

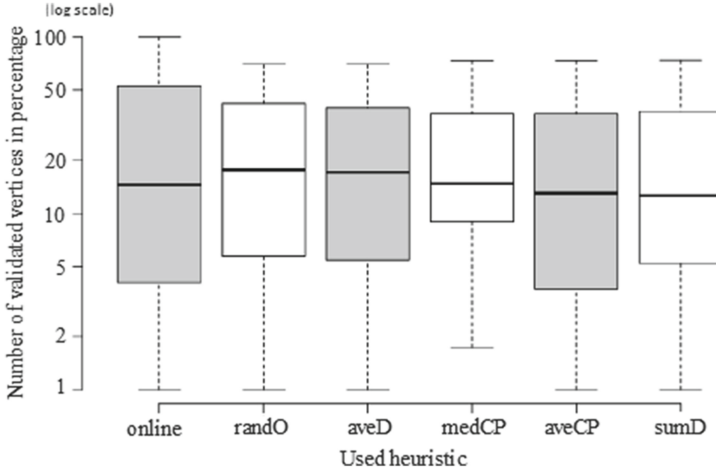

Fig. 5. Efficiency of our algorithm with online computed chain cover (online), minimum chain cover (randO), and heuristics determining the order in which individual chains are processed. The graph is in log scale.

cover, its minimality is guaranteed only before the first cutoff transformation is made as this transformation can shorten some chains of the cover and there can emerge some chains that can be joined together. The online heuristic always processes a chain that cannot be extended any more. It can thus possibly process even less chains than the original algorithm with the minimum chain cover precomputed. Moreover, the computation of the minimal chain cover is the most expensive operation of our algorithm besides the validation calls. The online heuristic does not need this precomputation and hence the $\sqsubset_G$ relation does not need to be computed. The time complexity of the algorithm is reduced to $\mathcal{O}(|V| + |E| + T_{valid} \cdot |C| \cdot \log L)$. We compare the online heuristic with the others in the next section.

3.5 Relaxing the Preconditions

The two main preconditions of our approach are that the graph is assumed to be acyclic and that the validation function is monotone on this graph. A natural question might arise whether we could relax one of these preconditions. Consider first an arbitrary annotated graph, i.e. a directed graph with a monotone validation function. The monotonicity implies that all vertices lying on one cycle are either all valid or all invalid. This means that we can preprocess the graph using any standard algorithm for decomposition into strongly connected components and work on the resulting (acyclic!) graph of strongly connected components.

Consider now a second possibility, where we retain the acyclic property of the graph yet relax the monotonicity precondition. If we run our algorithm on such a graph, we might not get the maximal valid cut of the graph. Nevertheless, the

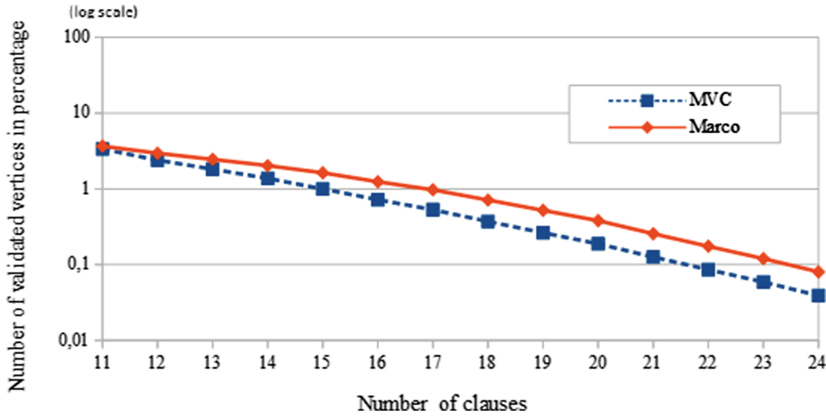

Fig. 6. A log scale graph that compares our algorithm with the Marco algorithm. The graph shows the percentage of subsets that were validated by the algorithms. Our algorithm is denoted by MVC (Maximum Valid Cut).

algorithm terminates and we obtain a set of vertices with the property that they are valid and their immediate successors in the graph are all invalid. We thus obtain at least partial evidence of the boundary between valid and invalid vertices. This can help us identify the source of errors in application areas such as software-product lines or in version control branches, which may not necessarily be monotone.

4 Experimental Evaluation

We implemented the cutoff algorithm and experimentally evaluated its behaviour on different types of graphs. While evaluating the algorithm we focused on the number of calls of the validation function as our aim is to minimise this number.

The first set of experiments was run on three different sets of randomly generated ADAGs of size up to 5000 vertices. The efficiency of the algorithm strongly depends on many factors like the relative number of pairwise unreachable vertices, the number and lengths of chains, density of the graph, etc. We tested the variant of our algorithm with online computed chain cover (online) and with precomputed minimum chain cover (randO). The results are shown in boxplot in Fig. 5, the boxplot shows the percentage of vertices which were validated. The online variant has higher third quartile but lower median.

Moreover, we tried the five heuristics described earlier. The heuristics sort chains from the minimum chain cover according to average cutting powers of individual chains (aveCP), medians of cutting powers of chains (medCP), average degrees of vertices of chains (aveD), and sum of the degrees of vertices of chains. The best performance was achieved using the sumD heuristic which has a median of 13 %.

Note that there are ADAGs for which almost all vertices have to be validated, namely graphs where almost all vertices are pairwise unreachable. These types of graphs were not included in our data sets for the experimental evaluation.

Requirements Checking. We now evaluate the performance of the algorithm on the graphs with the specific shape of a hypercube that represent all subsets of a set of requirements. For n requirements the hypercube consists of 2^n vertices and $n2^{n-1}$ edges. We used requirements specified in propositional logic and employed the SAT instances generator from [14] to generate experimental data. Experiments were run on requirements sets containing up to 24 requirements and hundred instances for each size. For these experiments we ran the online algorithm as it has shown to be the best one for hypercubes. The minimum-chain based approach performs worse on hypercubes as the minimal chain cover of a hypercube contains a large number of short chains. However, the binary search approach performs better on longer chains.

To provide a better insight into the qualitative parameters of our algorithm we compare its behaviour with two other tools solving the problem of finding the minimal unsatisfiable subsets of a set of requirements, namely [3,15]. Authors of [3] use the linear temporal logic (LTL) to specify requirements and report efficiency of around 10 % (i.e. 10 % of all vertices of the hypercube were validated). We were not able to repeat their experiments exactly as the authors do not provide their experimental data. Moreover, LTL is hard-coded in their tool. However, in our experiments with SAT instances the ratio of validated vertices decreases to 0.05 %. The MARCO tool, presented in [15], is proposed to solve any constraint sets. We compared the efficiency of our algorithm against MARCO on the same sets of SAT instances. As can be seen in Fig. 6, our tool makes less queries to the SAT-solver.

5 Conclusion

In this paper, we have focused on finding boundary elements in partially ordered sets, seen as a kind of graphs. We have discussed the mapping of this problem to various activities in software engineering; we have shown applications in safety and requirements analysis. We have presented a new general algorithm to solve this problem, including several variants and heuristics. We have found that the efficiency of the heuristics depends on the structure of the input graph. For graphs with the hypercube structure, the online variant of our algorithm performed the best.

As a future work, we consider several improvements of our basic algorithm. One possible direction of research is to aim at parallel processing of the configuration space in order to further improve the performance of our approach. Another is to focus more on the specific cases of hypercube graphs and exploit their structure more on the fly. We also want to consider more applications of our approach, such as software product line engineering and discovering incompatibilities in component-based designs. We also believe that our method can be

applied to various other domains, such as the parameter synthesis for biological systems [4]. We intend to explore these applications in more detail.

Acknowledgement. The research leading to these results has received funding from the European Unions Seventh Framework Program (FP7/2007-2013) for CRYSTAL Critical System Engineering Acceleration Joint Undertaking under grant agreement No. 332830 and from specific national programs and/or funding authorities.

References

1. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005)
2. Barnat, J., Bauch, P., Beneš, N., Brim, L., Beran, J., Kratochvíla, T.: Analysing sanity of requirements for avionics systems. Form. Aspects Comput. **28**(1), 45–63 (2016). doi:10.1007/s00165-015-0348-9
3. Barnat, J., Bauch, P., Brim, L.: Checking sanity of software requirements. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 48–62. Springer, Heidelberg (2012)
4. Barnat, J., Brim, L., Krejci, A., Streck, A., Safranek, D., Vejnar, M., Vejpustek, T.: On parameter synthesis by parallel model checking. IEEE/ACM Trans. Comput. Biol. Bioinform. **9**(3), 693–705 (2012)
5. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in temporal model checking. Form. Methods Syst. Des. **18**(2), 141–163 (2001)
6. Belov, A., Marques-Silva, J.: MUSer2: an efficient MUS extractor. J. Satisfiability Boolean Model. Comput. **8**, 123–128 (2012)
7. Chen, Y., Chen, Y.: On the decomposition of posets. In: 2012 International Conference on Computer Science Service System (CSSS), pp. 134–138 (2012)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
9. Dajani-Brown, S., Cofer, D., Hartmann, A.C., Pratt, T.W.: Formal modeling and analysis of an avionics triplex sensor voter. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 34–48. Springer, Heidelberg (2003)
10. Dilworth, R.P.: A decomposition theorem for partially ordered sets. Ann. Math. **51**(1), 161–166 (1950)
11. Fulkerson, D.R.: Note on Dilworth's decomposition theorem for partially ordered sets. Proc. Am. Math. Soc. **7**(4), 701–702 (1956)
12. Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., Margaria, T.: Software engineering and formal methods. Commun. ACM **51**, 54–59 (2008)
13. Joshi, A., Miller, S.P., Whalen, M., Heimdahl, M.P.: A proposal for model-based safety analysis. In: The 24th Digital Avionics Systems Conference, 2005. DASC 2005, vol. 2. IEEE (2005)
14. Lauria, M.: CNFgen formula generator. http://massimolauria.github.io/cnfgen/. Accessed 11 Jan 2016
15. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2016). http://link.springer.com/article/10.1007%2Fs10601-015-9183-0
16. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008)