Implementing and evaluating Dynamic Partial Order Reduction Using Probe Sets

Tom Bruintjes

ABSTRACT
Currently, one of the major problems in explicit state model checking is state space explosion; the number of states in modern software systems is growing so rapidly that even the most advanced computers no longer provide enough power to compute a complete state space. Kastenberg and Rensink have proposed a new algorithm to reduce state space. However there is no implementation and thus no knowledge on how it will perform in practice. This paper reports on a preliminary implementation and benchmark results of the algorithm, which is called Dynamic Partial Order Reduction Using Probe Sets (DROPS).

Keywords
Dynamic Partial Order Reduction, Model Checking, Probe Sets, State Space Explosion

1. INTRODUCTION
Model Checking \cite{2} is a powerful method to automatic verification of systems. The idea of model checking is to verify if a given property is satisfied in a model with a single click of a button. In our case we model a software system and check if it satisfies the properties we define, such as deadlock freedom.

One technique in software verification is generating all the reachable system states and search this state space for states the system should not be able to reach (e.g., states in which the system is in deadlock). This technique is called explicit state model checking. In modern systems, the number of states often grows exponentially resulting in vast amounts of states that even todays computers are unable to explore. Concurrent systems especially suffer from this phenomenon because of the many possible inter-leavings.

A way to tackle this problem is reducing the number of states. Numerous approaches have been developed using different techniques. One of them, Partial Order Reduction, is based on partial ordering. The general idea is that different inter-leavings can be equivalent, i.e., resulting in the same system behaviour. Not all possible paths have to be explored in this case.

Kastenberg and Rensink have proposed an entirely new Partial Order Reduction algorithm \cite{1} and also proved its correctness. In contrast to many Partial Order Reduction algorithms, this algorithm is based on a setting where states are represented by sets of so called entities rather than concurrent processes. Entities are an abstract concept and can represent virtually anything. In this paper, we will use natural numbers as entities but in more realistic settings entities could for example be primitive types such as integers, booleans and strings or object types as found in object oriented programming languages.

States are manipulated by actions, which can be thought of as rules that act on entities. Actions can Read, Delete, Create and Forbid sets of entities. Each action is actually a quadruple of sets of entities. To see if an action can be performed in a given state \( q \) each set in the quadruple will have to be compared with the entities in \( q \). If \( q \) contains all entities in Read and Delete, and does not contain any entity in both Create of Forbid, the action can be performed in \( q \).

Notation: We use Ent to denote the set of entities in a given state space and \( E_q \subseteq \text{Ent} \) to denote the set of entities in a given state \( q \). The set of actions that define the state space is denoted as Act. For a given action, we also want to be able to indicate each set of entities that are read, deleted, created and forbidden:

\[
R_a, D_a, C_a, N_a \subseteq \text{Ent}, \text{entities read by Action } a \in \text{Act}
\]

\[
D_a, C_a, N_a \subseteq \text{Ent}, \text{entities deleted by Action } a \in \text{Act}
\]

\[
C_a, N_a \subseteq \text{Ent}, \text{entities created by Action } a \in \text{Act}
\]

\[
N_a \subseteq \text{Ent}, \text{entities forbidden by Action } a \in \text{Act}
\]

A system can be captured in the entity based setting by transforming it to entities and actions. How this is done is beyond the scope of this paper, it depends on the type of system we are dealing with. We assume a given beginstate and a set of actions are available. We can build up the state space by looking for actions that can be performed in each state we have added to the state space, beginning with the beginstate that was given. By manipulation of actions, new sets of entities (states) are formed. This process iterates until no new states can be created.

Often, more than one action can be performed in a certain state. By choosing a real subset of all the actions that can be performed in a state, we can reduce the state space. The only problem is, that certain relevant information could be lost from the state space. To ensure no information is lost, this subset has to satisfy certain conditions. In this Partial Order Reduction algorithm the subset is called a Probe Set.

Although Dynamic Partial Order Reduction Using Probe Sets (DROPS) has theoretically been proven, Kastenberg and Rensink have not yet implemented it. Probe sets have been defined but a systematic procedure for constructing them is still missing. This means the algorithm can potentially reduce state spaces, but there is no practical data available that can verify this. In this paper we implement the algorithm to determine the reduction achieved by DROPS. We show how to select proper probe sets and benchmark DROPS against a naive state space generating algorithm.
2. BACKGROUND

This section gives a detailed description of how Partial Order Reduction Using Probe Sets works. We assume the system, for which we are going to compute a state space, has been transformed to entities and actions. That means we already have a given beginstate and a set of actions.

The first concepts that we have to get familiar with are stimulation and disabling between actions. An action is stimulated by another action if the first action creates at least one entity that the second action has to read or delete, or if it deletes entities that the second action has to create or forbid. Disabling means an action deletes entities that the following action has to read or delete, or if it creates entities that the following action has to create or forbid.

Definition 1 (stimulation and disabling). Let \( a, b \in \text{Act} \) be two different actions and \( R_a, C_a, D_a, N_a \) be the entities that are respectively read, created, deleted and forbidden by action \( a \) or \( b \). Stimulation \( (a \triangleright b) \) and disabling \( (a \triangleleft b) \) are defined as follows:

\[
\begin{align*}
    a \triangleright b & \Leftrightarrow C_a \cap (R_b \cup D_b) \neq \emptyset \lor D_a \cap (C_b \cup N_b) \neq \emptyset \\
    a \triangleleft b & \Leftrightarrow D_a \cap (R_b \cup D_b) \neq \emptyset \lor C_a \cap (C_b \cup N_b) \neq \emptyset
\end{align*}
\]

Example. We provide an example that could help the reader get a better understanding of stimulation and disabling. Table 1 shows a set of actions \((a, b, c, d)\) that apply to entities. As we can see, action \( c \) wants to read entity 2 and delete entity 1. Action \( b \) creates entities 1 and 2 so, action \( c \) is enabled by action \( b \). Action \( a \) also creates entity 1, but it does not create an entity 2. In order to enable action \( c \), entity 2 first has to be created. We say action \( a \) does not enable action \( c \) but it stimulates action \( b \), because it partially enables \( b \). In contrast to enabling/stimulating, an action can also be disabled by another action. In this example action \( c \) disables action \( d \), because entity 1 is deleted by action \( c \), while action \( d \) wants to read it.

<table>
<thead>
<tr>
<th>( R )</th>
<th>( D )</th>
<th>( C )</th>
<th>( N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td>1,2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( c )</td>
<td>2,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( d )</td>
<td>1,3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the proposed setting states are uniquely defined by their entities. We assume states consist of finite sets of entities. Because actions manipulate states we are basically dealing with transition systems labeled by \( \text{Act} \). Figure 1 shows an example of a (complete) state space which we will use in this paper to clarify certain statements. The corresponding actions can be found in Table 1. Mind that the actions in this example are named \( x^1 \) and \( y^1 \), while states are indicated by \( q_n \). The beginstate of this state space is \( (0,0,10) \).

Next, we define words: \( v,w \in \text{Act}^* \) as sequences of actions. If an action or word can be performed in a given state, we say it is enabled.

Definition 2. Let \( q \) be a state and \( w \) a word (or action in case the word consists of only one action). \( w \) is said to be enabled in state \( q \) if:

\[
    q \vdash w \Leftrightarrow \exists q' \text{ such that } q \rightarrow w
\]

Transition systems based on entities are deterministic, this means a sequence of actions leaving a state uniquely defines a target state. So-called vectors \((q,w)\) define both a target state \( q\uparrow w \) and a path leading to that state.

Definition 3 (Vectors). A vector \((q,w)\) consists of a state \( q \subseteq \text{Ent} \) and a word \( w \in \text{Act}^* \) such that \( q \vdash w \)

Definition 4. A vector \((q,w)\) characterizes a target state \( q\uparrow w \), i.e., the state we reach when \( w \) is performed in \( q \)

\[
    q\uparrow w := q' \text{ such that } q \rightarrow w
\]

This means vectors do not only capture their state, but also the history of actions leading up to that state. For example \( q_{06}(x^1 \cdot x^2 \cdot y^1) \) in Figure 1 characterizes state \( q_{06} \). The history of state \( q_{06} \) is thus said to be \((x^1 \cdot x^2 \cdot y^1)\).

Words can obviously be equal, but they can also be equal up to permutation (\( \equiv \)). Two words are equal up to permutation if they have the same length and contain the same action, although not necessarily in the same order.

We define two more relation over words, prefixes and a weak prefix. A prefix \( v \) of word \( w \) is equal to the first part of a word and a weak prefix \( v \) is equal up to permutation to part of some word \( w \).

Definition 5 (Prefixes). Let \( v,w \in \text{Act}^* \) be words, we define \( v \) as both a (strong) prefix and a weak prefix of \( w \)

\[
    v \preceq w \Leftrightarrow \exists u : (v \cdot u) = w \text{ (prefix)} \\
    v \preceq w \Leftrightarrow \exists u : (v \cdot u) \equiv w \text{ (weak prefix)}
\]
The weak difference between the words \( w \) and \( v \), is the remainder we have left when removing all the actions in \( v \) out of \( w \).

**Definition 6 (weak difference).** Let \( v, w \in \text{Act}^* \) be words, the weak difference between \( w \) and \( v \) is:

\[
w - v = u \text{ such that } v 
\]
function mapping enabled actions in a given state onto words: 
\[ \text{enabled}(q \uparrow w) \rightarrow \text{Act}, \text{denoted } p(a). \]

**Definition 14 (Probe sets).** Let \((q,w)\) be a vector and \(q\uparrow w\) its target state, we say that an action \(a\) is in the probe set \((a \in \text{dom}(p))\) if it satisfies the following conditions:

1. For all \(a \in \text{dom}(p)\) and \(b \in \text{enabled}(q \uparrow w)\), \(b \cdot a\) implies \(b \in \text{dom}(p)\)
2. For all \(a \in \text{dom}(p)\) and \(b \in \text{enabled}(q \uparrow w)\), \(p(a) \not\subseteq b\) \(w\) implies \(b \in \text{dom}(p)\)
3. For all \(a \in \text{dom}(p)\), \(p(a) \leq \downarrow_a w\)

The above definition states that if action \(a\) has been selected for probing and another action \(b \in \text{enabled}(q \uparrow w)\) disables \(a\), we have to include \(b\) in the probe set as well. The last two conditions state that if the part we discharge from the causal history conflicts with the prime cause of any other enabled action in \((q \uparrow w)\), the action with conflicting prime cause also has to be included. A detailed description of probe set construction is given in section 5.1

There is a trade-off between the size of the probe set and the length of the causal history. Minimizing the probe set results in a smaller causal history to keep track of; minimizing the causal history will result in a larger probe set and thus more actions and states to explore. The efficiency of the algorithm heavily depends on how the probe set is constructed.

The algorithm, as described so far, still suffers from one problem: starvation. When an action is not selected in the probe set its selection for probing could be postponed indefinitely. To overcome this problem, DROPS keeps track of the age of enabled actions \((\alpha : \text{Act} \rightarrow \mathbb{N})\). While constructing probe sets the algorithm has to ensure that at least one action with the maximum age \((\max \alpha)\) is included.

**Definition 15.** Let \(\alpha\) be a function that maps a certain age to actions \(a\) and \(b\). We say \(\text{dom}(\alpha)\) are all the actions for which such a map exists. The action with the maximum age is defined as follows:

\[
\max \alpha := a \in \text{dom}(\alpha) \mid \forall b \in \text{dom}(\alpha) : \alpha(b) \geq \alpha(a)
\]

### 2.4 Putting things together

Now that we can identify missed actions and select proper probe sets, we can describe the DROPS algorithm. Besides vectors we also need to keep track of the age function, therefore DROPS works with so called **Continuation Points**. Continuation points contain a state, its causal history (a vector) and the age function. Every action that is performed, missed or probed, creates a new continuation point. As long as there are continuation points DROPS will keep exploring. The full DROPS algorithm can be found in listing L1.

The algorithm initializes its state space (\(\text{STATES}\)) to empty and creates a continuation point for the beginstate, which is immediately added to the set of continuation points \(\text{CP}\) (line 2). After initialization, DROPS enters its main loop (line 3) and continues this loop while there are continuation points. An arbitrary continuation point is taken for exploration. If the state in this continuation point is already in the state space, we discharge it and continue with the next one. Otherwise, we add the state to the state space and determine all the missed actions (line 8).

---

**L1: Dynamic Partial Order Reduction Using Probe Sets**

1. \(\text{STATES} \leftarrow \emptyset\)
2. \(\text{CP} \leftarrow (\text{beginstate}, \epsilon, 0)\)
3. **while** \(\text{CP} \neq \emptyset\)
4. \(\text{take } (q,w,a) \in \text{CP}\)
5. \(\text{CP} \leftarrow \text{CP} \setminus ((q,w,a))\)
6. if \(q\uparrow w \not\in \text{STATES}\)
7. \(\text{STATES} \leftarrow \text{STATES} \cup \{q\uparrow w\}\)
8. **for all** \((v \cdot m) \in \text{dom}(q,w)\)
9. \(\text{STATES} \leftarrow \text{STATES} \cup \{q\uparrow v\} \mid v \leq v\)
10. \(\text{CP} \leftarrow \text{CP} \cup \{q\uparrow (v \cdot m), \epsilon, 0\}\)
11. **end for**
12. \(\text{construct probe set } \text{p} \text{ according to definition 14}\)
13. **update age function } \alpha\)
14. \(\text{CP} \leftarrow \text{CP} \cup \{q\uparrow p(\alpha), (w \cdot a) \rightarrow p(\alpha), a \mid a \in \text{dom}(p)\}\)
15. **end if**
16. **end while**

The state space is then restored by performing all the missed actions. This involves adding the intermediate states we come across when traversing the prime cause of the missed actions (line 9); and adding a new continuation point for the resulting state, where the causal history and aging function are empty again (line 10).

After we have dealt with the missed actions, we construct a probe set (line 12) for the current state. The probe set has to satisfy all three conditions of definition 14 and include at least one action from the age function \(\alpha\) with the maximum age. First \(\alpha\) is updated so all non-probed actions will be one cycle older in the next round. Actions that have become enabled in this round will be added to \(\alpha\) as well, with their age initialized to 0. Then for each action \(a\) in the probe set a new continuation point is added to \(\text{CP}\). These continuation points contain the next state, resulting from performing \(a\); a causal history, taking the weak difference between the causal history we received from the current continuation point and the sequence of actions defined by the probe set \((w \cdot a) \rightarrow p(\alpha)\). Finally the just updated function \(\alpha\) is provided to create the new continuation points.

After adding the new continuation points DROPS continues with the next cycle, taking another arbitrary continuation point.

### 2.5 Related Work

DROPS is not the first algorithm to reduce state space. Traditionally static approaches where taken for constructing stubborn [3], persistent [4] or amply [5] sets. Recently, dynamic approaches have been proposed such as in [6], where persistent sets are computed dynamically. For many algorithms, actions and variables have to be an a priori known. DROPS is based on a more abstract setting, entities can be created and deleted dynamically. This makes the algorithm suitable for dynamic systems where a priori bounds on actions or variables are not known.

Looking at the bigger picture of model checking, DROPS was developed (as an optimization) for model checking Graph Transformation Systems [7][8]. Graph Transformation Systems do not cope well with the established techniques such as persistent sets. The entity based setting can capture Graph Transformation Systems, meaning explicit state model checking can now be applied to Graph Transformation Systems.
3. PROBLEM STATEMENT

Explicit state model checking has been successful in verifying concurrent software systems. There is however the problem that today's systems result in state spaces even modern computers cannot handle. Potentially, DROPS reduces the size of the state space without excluding any system behavior. In theory the algorithm works, however it needs to be tested in practice before we can come to conclusions.

We know certain probe sets exist according to definition 14, however a systematic procedure for construction is not given by Kastenberg and Rensink. Implementing the algorithm requires us to develop a way of constructing probe sets. The efficiency of DROPS depends on the construction of probe set so we will need good heuristics to select the right actions for the probe set.

We intend to get some insight in the performance of DROPS by implementing the algorithm. In order to do so, we will provide a systematic procedure for construction of probe sets, based on heuristics. After DROPS is implemented, we want to compare it against an algorithm that computes the state space without any reduction at all. Such an algorithm does not yet exist, because we are dealing with an entirely new approach. This means we need a second state space generating algorithm that can generate complete state spaces. We will develop this algorithm ourselves. By comparing the results of both algorithms, we can determine how much state space reduction we can achieve and how much faster or slower DROPS is compared to a naive state space generation algorithm.

In the remainder of this paper we will therefore focus on the following questions:

- How much reduction can we achieve with DROPS?
- How much time do we win or lose with DROPS compared to a naive computation?
- What are useful heuristics for constructing probe sets?

4. APPROACH

We have an algorithm (DROPS) to reduce state spaces, that has theoretically been proven correct. However, there is no practical knowledge available so we are unable to tell how well DROPS can reduce state spaces. The aim of this paper is to implement DROPS, and find out how useful it is in practice based on some examples. For a Partial Order Reduction algorithm to be useful, we want it to be able to substantially reduce the state space. Furthermore, we want the time it takes the algorithm to compute the reduced state space to be less or reasonably close to the time it takes a naive algorithm to compute the state space.

Like our example in section 2, entities are represented by natural numbers. State spaces are predefined by a beginstate and a set of actions. Most parts of the algorithm are directly translated from their corresponding definitions of section 2. We deviate a bit from the original algorithm by Kastenberg and Rensink: edges are explicitly stored. This makes it easier to determine the amount of reduction in edges as well as states.

For probe set construction we develop a systematic procedure based on heuristics. This optimizes the probe set specifically for the type of state space we use throughout this paper, which are those that are diamond or triangular shaped (depicted in Figure 2). Because we can not say anything about the performance of this probe set in other state spaces, we base our results only on these two types.

4.1 Naive complete state space generation

DROPS is an entirely new approach, so there is no material we can use as a reference for benchmarking. In order to be able to say something about the efficiency of DROPS, we need a usual algorithm that can compute state spaces based on entities. This algorithm must generate a complete state space in contrast to DROPS. Listing L2 shows the algorithm that we use for this purpose. As the name might give away, naive state space generation is a very simple algorithm. Every time an action becomes enabled we perform it, expending the state space. The algorithm continues until there are no actions left to perform.

```
L2: Naive state space generation
1. VISITED ← ∅
2. EXPLORE ← beginstate
3. while EXPLORE ≠ ∅
4.   take q ∈ EXPLORE
5.   if q ∉ VISITED
6.     VISITED ← VISITED ∪ {q}
7.     for each a ∈ q ⊢ a
8.       EXPLORE ← EXPLORE ∪ {q[a]}
9.   end for
10. end if
11. end while
```

In naive state space generation, we maintain two sets of states: one set of states we have visited (VISITED) and one set we have only 'seen' (EXPLORE) by performing actions. The first set of states will eventually contain the full state space. The second temporarily stores the states that are found, when performing enabled actions in a given state (line 8).

The state space computation is very straightforward. We begin by adding the beginstate, which is assumed to be given for each state space, to the EXPLORE set. The algorithm then enters the main loop that continues until there are no more states to explore.

Figure 2: Diamond and Triangular shaped state spaces

State spaces can be differently shaped. In Figure 2, for example, we find two different shaped state spaces. A diamond shaped state space, and a triangular shaped state space. Another advantage of DROPS over approaches using persistent sets is that reduction based on probe sets is independent of the shape of a state space [1]. Algorithms based on persistent sets results in little to no reduction at all in triangular state space.
In every cycle, an arbitrary state \( q \) is taken from EXPLORE. If we have not been to this state \( (q \notin VISITED) \), we add it to VISITED. Otherwise, we discard it to prevent re-exploring. Then, for each action \( a \) that is enabled in \( q \), we determine the target state \( q \uparrow a \). This target state is then added to EXPLORE.

Although, not explicitly listed we also store edges between states. When performing action \( a \) in state \( q \), we can simply store an edge from \( q \) to \( q \uparrow a \). Unlike DROPS, storing edges is required in this algorithm since they are not implicitly stored in a history.

4.2 Benchmarking
For benchmarking we need some point of reference. This is where we run into another issue, caused by the new entity setting. Dynamic Partial Order Reduction in this context has not yet been attempted. Comparing results with existing techniques such as persistent sets would not be very meaningful, because of the completely different approach. We have to create our own point of reference. The benchmark for this paper consists of multiple predefined state spaces based on entities. As we mentioned before, we look at two different kinds of state spaces: triangular shaped and diamond shaped.

Now that we have a point of reference, we have to make clear that there are two kinds of reduction we can look at. DROPS does not only store states and edges but also a path (causal history) for each state. This means we can look at absolute reduction, meaning less states and/or edges. Or we could look at the total reduction expressed as the amount of memory used by the algorithms.

The final step is to let both algorithms compute the same state spaces and determine the difference between the results. The difference will tell us how much reduction we can achieve with DROPS. We can also measure the time it takes both algorithms to complete a full run. However, this is only useful for substantially large state spaces. If the examples are small, computation will not take long, so overhead such as class loading will have too much influence on the measurements.

5. RESULTS
The results from benchmarking DROPS show a satisfying amount of reduction for both triangular and diamond shaped state spaces. An overview of the data extracted from our benchmarks can be found in section 5.2. With our preliminary implementation we can now conclude that, at least for triangular and diamond shaped state spaces, Partial Order Reduction Using Probe Sets works as we had hoped. We expect to see similar results in other types of state spaces.

The results of these early experiments look promising but the current implementation is not yet optimal. We have not been successful in implementing the part where we discharge part of the causal history. This means a penalty for the performance. Since we are unable to discharge anything, the causal history keeps growing. This technical issue made us decide to look at absolute reduction instead of measuring memory usage. However, absolute reduction is also degraded somewhat. In essence, property 2 and 3 of definition 14 say that whenever the prime cause of action \( a \in \text{dom}(p) \), conflicts with another enabled action \( b \), we have to include this action \( b \) in the probe set as well.

Let \( a, b \in \text{Act} \) be actions that are enabled in vector \( (q,w) \). \( w \) is conflicting with \( a \) if \( \downarrow_a w \cap \downarrow_b w ) \neq \emptyset \). As we descend deeper into a state space, the prime cause of each action is likely to become larger. This means we are less likely to successfully leave enabled action out of the probe set because of conflicts. The result is that the probe sets become larger, so we achieve less reduction. Fortunately, this effect is not an issue in smaller state spaces. Therefore we limit ourselves in this paper to small state spaces such as the one in Figure 4. This also means that for the time being, we can not conduct time measurements. Overhead would cause too much interference.

5.1 Constructing probe sets
We explained in section 2.3 that our probe sets are actually a partial functions. Within each state, we select a proper subset of enabled actions. These actions are mapped to a sequence of actions that are a weak prefix of the state’s causal history. Each state gives rise to a set of possible probe sets according to definition 14. As a consequence there are two ways to obtain a suitable probe set. One way is to compute all possible probe sets and select the one that has the right properties; the other is to use a general, but smart, algorithm for construction of a probe set. The first method requires many resources and would probably defeat the gain of DROPS. We believe that a smart algorithm based on heuristics is the best way to obtain probe sets.

When constructing probe sets, we have to keep in mind there are two sides that have to be optimized. On the one hand we would like to keep the number of actions as small as possible, on the other hand we have to minimize the part of causal history that we have to remember. There is not one general optimum, so heuristics are needed to get us as close as possible to the best performing configuration.

Constructing probe sets can be relatively easy if we translate the properties of definition 14 into an algorithm. All we have to do is choose one action to begin with and keep adding actions until we satisfy all three conditions. Instead of selecting the first action at random we rather select this action based on a heuristic, helping us to minimize the probe set and causal history. In this implementation, we select the first action based on the length of its prime cause. Whether the action with the longest prime cause or the one with the shortest prime cause is chosen depends on on the kind of state space we are trying to reduce.

For diamond shaped state spaces the action with the longest prime cause appears to be most effective. This action tends to be the one that leads to the outer side of the state space, causing the algorithm to leave out states and edges on the inner part of the state space. At the same time this action ensures us that the part of history we can discharge is fairly large as well. The idea is illustrated in Figure 4. In the example of section 2.3 we have taken the actions with the longest prime cause \( (x^1 \cdot x^2) \), \( (x^3 \cdot x^2 \cdot x^4) \) and ignored \( y^1 \). By doing so, the middle state is not included in the reduced state space. If the right side of the state space is computed in the same way, the reduction would be 1 state and 4 edges.

For triangular shaped state spaces reduction does not seem to really benefit from either choosing the longest or the shortest prime cause. Both methods perform equally well. Section 5.2 contains an overview of the achieved reduction by DROPS. These result are based on the probe set that delivers the most reduction in that specific state space. We have listed the steps our current algorithm takes when constructing probe sets below.
1. Start of by adding one action with the maximum age in \( \alpha \) to the probe set.

2. For each action that is currently enabled determine the prime cause.

3. Add the action that has the longest/shortest prime cause, if the probe set does not already contain this action.

4. Determine all the actions that have a prime cause that conflicts with the prime causes of the actions already in the probe set.

5. Add all the actions that have a conflicting prime cause.

### 5.2 Reduction performance

In the beginning of this section we mentioned that because of technical issues, we are not able to measure the full potential of DROPS. The causal history grows unbounded, defeating the true purpose of the algorithm. However, we can still look at absolute reduction. Even with the hefty penalty we pay, DROPS manages to reduce state space quite well. Table 3 and Figure 4 show the results we obtained, using some small state spaces.

In Figure 4 we find the state space from our example in section 2 and a triangular state space. The first two images at the top show what the transition systems look like after a complete state space generation. The images at the bottom show transition systems of the reduced state spaces, computed by DROPS. Table 3 gives an overview of the reduction achieved by DROPS based on these two examples. The first columns show the amount of states and edges in the complete state space. The next two columns show the amount of states and edges after reduction, and the last column indicates the amount of reduction by DROPS per state space. States and edges are considered equally important in our analysis.

<table>
<thead>
<tr>
<th></th>
<th>Complete</th>
<th>Reduced</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>8</td>
<td>6</td>
<td>35%</td>
</tr>
<tr>
<td>Edges</td>
<td>12</td>
<td>7</td>
<td>35%</td>
</tr>
<tr>
<td>Diamond</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle</td>
<td>9</td>
<td>8</td>
<td>20%</td>
</tr>
</tbody>
</table>

### 6. CONCLUSION

Kastenberg and Rensink have developed a new Partial Order Reduction algorithm. The advantages of this algorithm (section 2.5) include: reduction of new types of systems, e.g., graph transformation systems; Reduction is independent of the type of state space and state spaces can be computed dynamically, meaning actions do not need to be a priori known.

We believe these advantages alone are reason enough to put more effort in this algorithm. Even while our results are not optimal we can see potential in DROPS.

We have also shown that it is not that difficult to construct probe sets. The approach we provided is based on the length of the prime cause of the first action. Similar heuristics can be used in different approaches. It is a matter of experimentation to find out what further heuristics are also useful. Especially for triangular state spaces the probe set needs further tuning.

### 6.2 Future work

There is still much work left. The first task at hand is to repair the preliminary implementation so we can discharge parts of the causal history. Section 6 discusses a possible approach to solve the problems in the current implementation. As soon as this problem is solved, we can benchmark DROPS with large state spaces. This is where DROPS will really prove itself. We could then also measure the time it takes both algorithm for a state space computation, and finally decide if DROPS is worth the effort.
If DROPS proves to be worthwhile in larger state spaces, a full scale implementation can be realized. Eventually Kastenberg and Rensink would like to see DROPS integrated in the tool GROOVE [9].

Another point of improvement is the construction probe sets. We have provided just one heuristic. Some experimentation with other heuristics may prove to be useful. Especially since triangular shaped state spaces do not really seem to benefit from either selecting the action with the shortest or the longest prime cause first.

Regardless of all the work that still needs to be done, we think the initial results with DROPS look promising. In our opinion, it would be worthwhile to investigate further into this new approach of state space reduction.

7. DISCUSSION
We already mentioned more than once that because of technical reasons we have been unable to achieve the full potential of DROPS. Currently, an unidentified issue prevents us from discharging part of the causal history. This causes the algorithm to be less efficient. We have looked into the problem and provide a possible cause here.

On the level of implementation, vectors can be interpreted in two ways. An object consisting of a state (Set of entities) and a path (List of actions) leaving this state; Or a path leading up to a state. In the early stages, we chose to implement the algorithm based on the second interpretation. Concepts such as over-approximation and backtracking are completely depending on this causal history. The problem we are dealing with, is that deleting part of this history results in lossy reduction. Certain missed actions are not detected anymore when parts of the history are removed. We suspect that for our implementation too much information is lost from the causal history in order to perform a correct missed action analysis. A possible solution could be to rewrite the implementation based on the first interpretation mentioned above.

Acknowledgments
Our thanks go out to Harmen Kastenberg, Marielle Stoelinga, Lodewijk Bergmans, Harold Bruintjes and fellow students.

REFERENCES