Combining Structural Reductions with Partitioning

Martin Christensen

April 28 2021

Chapter 1

Introduction

This paper deals with the possibility of using structural reductions and partitioning of the state space when checking reachability. It is inspired by a method called Compositional Backwards Reachability (CBR) which was first introduced in [4]. The CBR paper mentions functions for partitioning and heuristics for which partition to explore first as areas for future research which is what we're examining.

We examine the possibility of partitioning the state space such that certain edges are unavailable in one partition but available in another, in a way that would ideally create at least one greatly simplified model which can be verified very quickly. We combine this approach with structural reductions applied seperately to the partitions with the goal of selecting partitions that can be greatly reduced as our heuristic.

Model Checking

Model checking is a topic in computer science where a model of a system is used to answer properties about a given model, such as whether a state in the model is reachable or if the system contains deadlocks. It is used among other things in industrial systems for examining setups of industrial robots and internet protocols to check that distributed systems behave as intended.

Model checking[4] is done by exploring the model for a given query where a starting state is given and rechable states from that state are gradually added to a passed list until either a state matches the query or there are no remaining reachable states that have not already been explored.

1.0.1 State Space Explosion Problem

As a model increases linearly in size the state space increases exponentially. This is known as the state space explosion problem. Many papers address this using different techniques like zones[2] for timed models, structural reductions or stubborn sets[1], but it is the key limiting factor in model checking as ex-

ponential complexity greatly limits how large models can be before the time it takes to answer a query becomes impractical.

It is also a problem in terms of memory usage as the passed list eventually grows too large to keep in memory.

1.0.2 Compositional Backwards Reachability

Compositional Backwards Reachability CBR is a method where the reachability search is performed backwards, from the reachable state to the initial configuration, allowing faster exploration via depth first search on models that support the approach. Another advantage is that exploring a single step forward every state produces exactly one successor state per edge, but for edges that set variables a single state can have many predecessors, letting CBR explore multiple states in one step. It is described in greater detail in [4].

Compositionality allows the search to be performed over partitions in the search space, for example by first exploring only some of the automata in a composition. This does however require that the ignored partitions are unable to influence the partition we are examining, such as by preventing delays in timed systems or inhibitor arcs from petri nets.

For example the setup in figure 1.1 where we select automata A and B as our partition. If we explore using only the partition we get a positive answer for our query, but as automata C will enter a section that prevents delays due to the invariant it is not actually reachable in the full model.

1.0.3 Structural Reductions

Structural reductions described in [2] is a technique for reducing the size of a model by removing parts of it in a way that does not affect the outcome of a query. For example in Figure 1.2 we can see this applied in a trivial case.

Removing a single state from a model can have a large increase in performance for highly parallel models. If we have two parallel automata from our basic example we have a total of 3^2 states; $\{a,a\},\{a,b\},\{a,c\},\{b,a\},\{b,b\},\{b,c\},\{c,a\},\{c,b\},\{c,c\}$ whereas the reduced automata only have 2^2: $\{a,a\},\{a,c\},\{c,a\},\{c,c\}$. Performing reductions on models is also typically much faster than exploring them, except for cases where the query is trivial to answer like in a single step.

As with CBR there are limits to when they can be applied. Structural reductions are not deadlock preserving, and thus when applied to timed models where it is critical at which times a location is accessible fewer reductions can be applied, as demonstrated in figure 1.3.

Generally the more intricate the semantics of the system is, the harder it becomes to perform structural reductions.

Structural reductions have primarily been applied to petri nets but we believe this is mostly due to the types of problems that are natural to model in petri nets; highly parallel systems where the processes have no internal impact on each other. This problem area naturally meets the requirements for both structural reductions and CBR, but one major limitation of petri nets is that they are

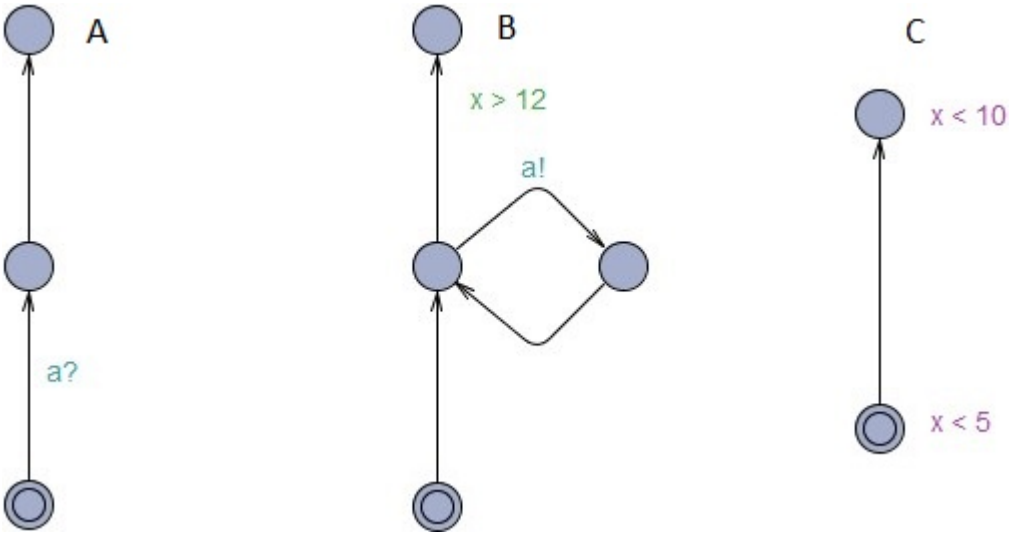

Figure 1.1: An example of inappropriate partitioning for a set of Timed Automata with the semantics of the Uppaal model checker. If we try to explore A and B without also exploring C we would find that the end locations for each process are reachable, yet since C would prevent us from delaying more than 10 time units this would not actually be true.

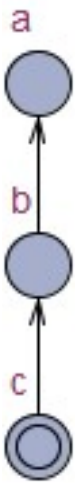

Figure 1.2: Example of a basic reduction. This simple model starting in c where it can only move to b then a can have the b location removed so we simply go straight to a as long as b is not a part of the query.

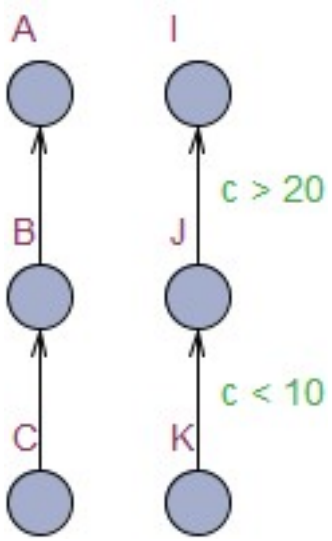

Figure 1.3: Timed vs Untimed reduction. While we can safely remove location B, we cannot remove location J and preserve the behavior of the model, as K must become unavailable at 10 time units, and I cannot become available before 20 time units.

not used for composition of parallel automata which is a major application for model checking when checking compatibility between modular systems. We believe this is an area where structural reductions could potentially prove highly beneficial.

1.0.4 Combination Example

To give an example of how these two concepts can be used together, we consider a single type of reduction which removes only locations with a single ingoing edge and a single outgoing edge. We combine this with partitioning that takes a place with two outgoing edges and creates two seperate models; one in which the first edge is taken and one in which the second is taken, then we allow each partition to move to the other partition when in that location.

Consider the example in Figure 1.4

We have our initial model first prior to any partitioning and reductions. We first perform the reduction step, but since this model has no matching pattern; every location has multiple ingoing or outgoing edges, we cannot reduce anything.

We then partition the model into two new graphs; the top one being where we always perform $A \longrightarrow B$ and the bottom one always $A \longrightarrow H$. We do this by copying the original branch from the branching location only copying the locations connected via edges, until we reach the branch location again. This

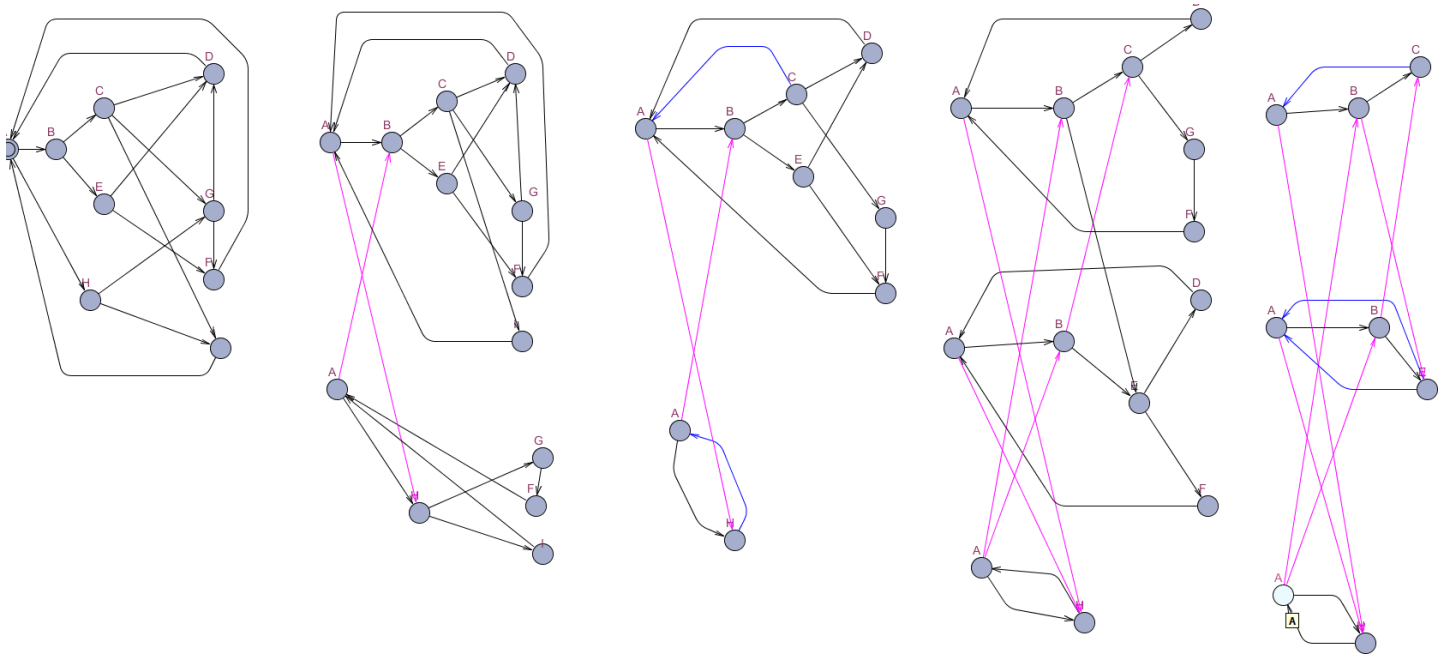

Figure 1.4: Example of a combined partitioning and reduction technique. Purple lines mark transfers between partitions and blue lines are reduced paths

creates two new partitions missing some locations and edges which are now only present in the other partition, and new purple edges leading between them.

Since we have removed connections from the graph we can now apply reductions again, this time eliminating location I in the top partition and locations G, F and I in the bottom. Reduced paths marked with blue.

For step 4 we again partition this time on location B, producing two new partitions one in which we always choose C in B and one where we always choose E. Step 5 we again apply reductions which leaves us with three much smaller partitions than the initial model.

Note that we can still eliminate duplicates across partitions, as every reachable state from any location in each partition is the same across partitions. This means that even though the model has 8 locations only 5 of them are unique.

This has the potential to double the model's size, but wont increase the state space, as the new locations added are, for the sake of states, identical to existing ones. This does however mean that as structural reductions are based on model size which is normally tied to state space, become more expensive to apply with more partitions relative to exploration, which is tied to state space. In addition, since partitions only contain a portion of the total state space, intermediate states removed via structural reductions may be reachable in another partition, which also reduces efficiency of reductions.

To the best of our knowledge optimizing structural reductions has not been examinined before as they are much faster than the exploration step, but with partitioning we now have the expectation that each partitioning at worst doubles the model size leaving exploration complexity untouched, but doubling the time it takes to apply reductions, thus reduction performance is essential to evaluating the algorithm. Because of this we assume that at some point the cost of exploring the model will be lower than what is gained by continually reducing it.

The goal of the combined algorithm is to partition the model in a way that cases preventing reductions are constricted to one of the partitions, enabling us to apply further reductions in at least one of them. We ultimately wish to partition the state space such that compositions that are highly reduceable, and thus presumed to be very fast to explore, are explored first.

Chapter 2

Problem Analysis

In this chapter we go into details regarding the challenges of combining structural reductions with partitioning and the motivation for the algorithm.

2.1 Motivation

This section describes the expected benefits of the proposed algorithm.

2.1.1 Reductions vs Techniques

A lot of techniques exist for speeding up exploration, such as partial order reductions [1], which solve the same problems as reductions, but because they are applied mid exploration they are run repeatedly when answering a query while reductions are only run once as preprocessing.

In figure 2.1 partial order reductions are applied for every state, while structural reductions are only applied once. This means partial order reduction might never be applied if no state is reached that warrants it, or it may be repeated millions of times. Generally runtime techniques have better best case scenarios while reductions are best in worst case as it is a hard reduction of the total state space.

2.1.2 Reductions in Parallel Automata

For petri nets reductions perform extremely well as they're commonly used to model highly concurrent processes. We believe the benefit of structural reductions is more closely tied to the kind of problems petri nets are naturally used to model rather than the petri net semantics themselves.

However there are already existing optimizations that overlap with reductions, and especially for parallel automata symmetri reductions are very efficient.[1] Symmetri redutions are shown to cause very high increases in performance [3] and structural reductions tackle the same problem of many parallel, identical automata.

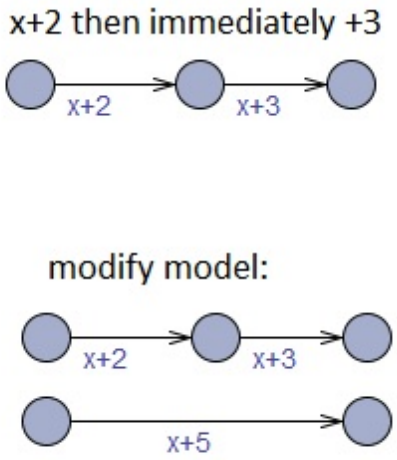

Figure 2.1: Partial Order is done during exploration, by combining the first and second update into one exploration step the explorer can skip the intermediate state saving memory. Just like the reduced example, both will produce a single successor state with $x + 5$.

Symmetri Reductions change the way states are represented for many identical automata, such that it tracks how many automata are in a specific state. This means for example with 3 automata and two locations each instead of having $1 : a, 2 : a, 3 : b, 1 : a, 2 : b, 3 : a, 1 : b, 2 : a, 3 : a$ we represent all three as $a : 2, b : 1$.

2.1.3 Distributed Model Checker

Another advantage to structural reductions is that they are completely independent of other parts of the model, which makes the reduction step easy to parallelize. In distributed exploration it's necessary to have a way to handle duplicate checking for reachable states across nodes, but for structural reductions you can just send a partition to each worker node and it is dependent on nothing else.

It is even possible to perform reductions and exploration simultaneously, letting one worker explore while the rest reduce untouched partitions, although this can lead to some states being explored that would've later been reduced, it still allows for early termination if an answer is found before the remaining partitions are removed through compositional backwards reachability [4].

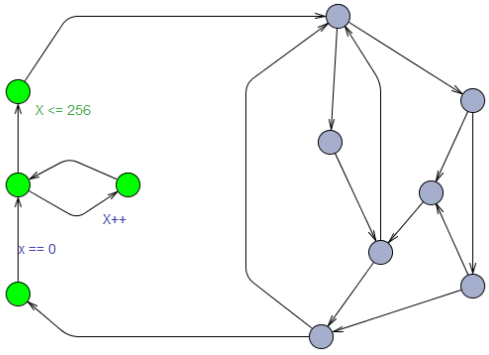

Figure 2.2: Fragmentation Problem. The variable sensitive section is marked in green.

2.1.4 Preprocessing

Some organizations keep models in storage for example component interfaces that are reused in new products. Structural reductions can be applied to existing models and compositions, saving reduced models to provide faster model checking on new queries. This does require user defined significant locations, the only locations accessible via query, in order to be useful, as the models may be reduced to nothing without a query preventing locations from being discarded.

2.1.5 Fragmentation Problem

The fragmentation problem deals with localized variables in models. Consider the example in figure 2.2

Here the variable is only accessed within a small part of the model, and is reset when the part is re-entered. This is a potential case for control flow models in particular, where we only access the variable for controlling components that are not frequently in use.

Active clock reduction is a technique used in uppaal[1] which allows the variable, or in this case clock, to be removed from the states. This is very potent as it is often possible to exit the section accessing the variable with many different values for the variable with everything else remaining the same which results in many unique states that all produce unique successor states only because of the difference in the local variable. This can multiply the reachable state space by every possible permutation of the variable in question.

A way to incorporate this using structural reductions could be to list the relevant variables in a partition, and ignore unlisted variables while checking for duplicates.

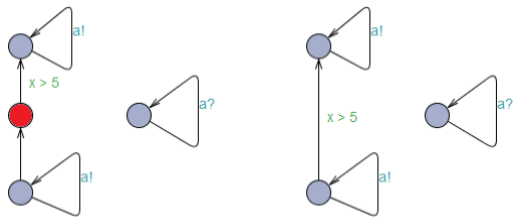

Figure 2.3: Partial Order is done during exploration, by combining the first and second update into one exploration step the explorer can skip the intermediate state saving memory. Just like the reduced example, both will produce a single successor state with $x + 5$.

2.2 Limitations

2.2.1 Deadlocks

one of the limitations in structural reductions is that they only preserve reachability, not deadlocks.

In the figure 2.3 the intermediate red location is safe to remove, it won't change what states the system is able to reach, however before reducing the model is able to deadlock by advancing to the red state when x is ¡= 5. The reduced model can never deadlock

Timing provides ample opportunities for deadlocking as a deadlock occurring somewhere else in the system affects which states are available. Same for global variables.

This means that we cannot use structural reductions for answering a deadlock query.

2.2.2 Parallel Automata

In models with many identical, parallel automata structural reductions look highly rewarding as removing a state in a duplicate automaton counts as multiple for each duplicate, so we would expect a 2^n reduction in statespace when applying it, where n is the number of parallel duplicates.

However due to optimization techniques like symmetri reductions described in (reference uppaal paper) in which rather than keeping track of each individual state of parallel automata, we instead count the number of automata in each state. Combined with this technique we would not see the 2^n gain but rather $nlogn$ gain, although this should still be quite good.

2.2.3 Antisynergy

If we can double the speed of model checking with reductions and double the speed with exploration techniques do we get $4x$ the performance? Sadly, no. Re-

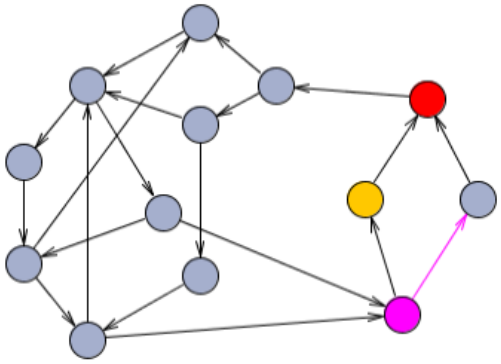

Figure 2.4: Example of a bad edge split. Here we split from the purple location but immediately rejoin in the red location, giving us only a single unique successor for each partition.

ductions and techniques often eat into the same mathematical principles, which means when used in conjunction they cannibalise each other's gains. As described with symmetri reductions, partial order reductions and path reductions also rely on the same principles and reduce each other's performance. While this is not always the case, it is not enough to show that structural reductions improves the answer time, it has to do better than existing techniques or show synergy where reductions + techniques outperform just techniques.

2.3 Partitioning

When we partition we duplicate the model while allowing exploration across them. This is different from the CBR approach which aims to partition the state space such that every state belongs to a unique partition. Our approach aims to split the model such that more reduction conditions can be satisfied. This means we have states occuring in multiple partitions from different predecessors, a property that is examined more closely in chapter 6

2.3.1 Edge split vs guard split

An edge split means from some location we choose one of multiple outgoing edges and direct it to another partition, while the successor locations from the original model. As shown in the partition overreach problem example this can lead to situations where each partition eliminates non-overlapping intermediate states. This problem occurs when e1 and e2 converge quickly after the partition.

We refer to the number of steps x in both partitions, $s_{start} \longrightarrow_x s_{shared}$, as the unique successor value, where s_{start} is the state before the split into

seperate partitions and s_{shared} is the average successor state encountered in both partitions. The problem is more pronounced when x is low, .

As seen in the example a bad split with low unique successor value occurs when the partitions immediately rejoin.

A guard split, where a single edge is split into two edges leading to the same location in seperate partitions more closely partitions the state space rather than the physical model. Since the states reachable initially in each partition differ on at least one variable, only variable updates are capable of producing shared successor states. A guard split simply needs to happen with no immediate updates to the variable used to partition. However, unlike the edge split, there is no guarantee that an arbitrary guard split will lead to any reductions, unlike edge split which will always reduce the number of ingoing edges in at least one location.

For this reason it is important to develop and test a good heuristic for where to split, especially for guard partitions.

Chapter 3

Problem

This section specifies the semantics and problem for the paper. As explained during introduction this algorithm is intended for use with highly parallel systems, however this paper is concerned with exploring the feasibility of such an algorithm first. Thus our semantics are kept within very limited functionality only considering basic variable manipulation without parallel automata or timing to focus on the reductions and partitioning of a single process.

3.0.1 Preliminaries

A Simple Integer Automaton P is a tuple $P = (L, L^{init}, L^{crit}, E, V, V_{min}, V_{max})$ where

L is the set of locations. L^{init} is the initial location and nonempty. $L^{crit} =\subseteq L$ is the set of critical locations and $L^{init} \in L^{crit}$

E is the set of edges defined as $E \subseteq L \times G \times U \times G \times L$

G is the powerset of guards: $G = \mathcal{P}(g)$

min and max are the minimum and maximum values variables are allowed to take. These are needed to keep the state space finite in case of loops.

Guards

A guard g is defined as a lower and an upper bound, g_l, g_u such that $0 \leq g_l \leq g_u \leq max$ All edges have guards for every variable, if no guard is specified then the guard is $\{min, max\}$.

A valuation satisfies a guard, $v \vdash g \iff g_l \leq v \leq g_u$

Domination For a single variable we define domination as $g \subseteq h \iff g_l \geq h_l$ and $g_u \leq h_u$

Non-intersecting Two guard sets are Non-intersecting, $G \times H$ if $g_l > h_u$ or $h_l > g_u$

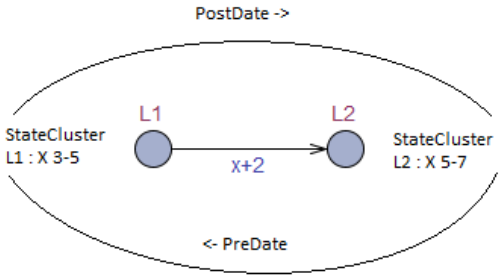

Figure 3.1: Predate and Postdate in relation to the original guard. Think of these as copying a guard forwards or backwards one edge in a path, modifying them by the updates they cross, such that the path will permit the same states to travel across.

intersection Guard intersection specifies the set of guards that will allow states allowed by both sets of guards:$J = G \cap H$

We first define the guard intersection for a single variable:

$g \cap h = j$ where

$$G_1 \cap G_2 = G' = \left\{ \begin{array}{ll} j_l = max(g_l, h_l), j_u = min(g_u, h_u) & \text{for } g_l \leq h_u \text{and} g_u \geq h_l \\ \emptyset & \text{for } else \end{array} \right\}$$

$G \cap H = J | \forall j \in J. \forall v \in V. j_v = g_v \cap h_v$

Guard Transformations are used during reductions to create guards that are satisfied by the predecessor or successor states in relation to an update. For example when defining the guard of a new edge that's the combined edge of taking e_1 then e_2 we would need to allow values that satisfy the guard for e_2 after applying the update of e_1.

these are:

Predated Guard $g \triangleleft_u = g'$ is the guard g' that accepts the values that are accepted by g `before` applying update u such that$v \vdash g \longrightarrow u(v) \vdash g'$ It is defined as

$$g \triangleleft_u = g' = \left\{ \begin{array}{ll} max(V_{min}, u(g_l), min(V_{max}, u(g_u)) & \text{for } u_\pm \in \{"+=", "-="\} \\ min, max & \text{for } u_\pm = "=", g_1 \leq u_k \leq g_u \\ \emptyset & \text{for } u_\pm = "=", u_k < g_l \\ \emptyset & \text{for } u_\pm = "=", u_k > g_u \end{array} \right\}$$

where $u_v = y_v$

Postdated Guard $g \triangleright_u = g'$ is the guard g' that is satisfied by the values satisfying g when updated by u `after` g such that $v \vdash g' \longrightarrow u(v) \vdash g$ It is defined as . $g \triangleleft_u = g' = \left\{ \begin{array}{ll} max(V_{min}, u^{-1}(g_l), min(V_{max}, u^{-1}(g_u)) & \text{for } u_\pm \in \{"+=", "-="\} \\ u_k, u_k & \text{for } u_\pm = "=", g_1 \leq u_k \leq g_u \\ \emptyset & \text{for } u_\pm = "=", u_k < g_l \\ \emptyset & \text{for } u_\pm = "=", u_k > g_u \end{array} \right\}$

where $u_v = y_v$

Updates

U is the powerset of updates $U = \mathcal{P}(u)$

An update $u = u_v, u_\pm, u_k | u_\pm \in \{=, + =, - =\}$, $v \in V$ and k is a constant.

The inverse update u^{-1} swaps plus with minus and is unchanged for equality.

Upmerge : An upmerge $U' = U \bigoplus Y$ is a an update applied to the outcome of another update; applying U then Y. For example, if the first update is $+ = 3$ and the second is $+ = 5$ we can combine it into one update of $+ = 8$, while if it was $+ = 3$ followed by $= 2$ we skip the first as its overwritten. We first define this for a single variable:

$$u \bigoplus y = u' = \begin{cases} u' = y & \text{for } y \\ u' = u_v, " = ", u_k + y_k & \text{for } u_\pm = '=', y_\pm = \\ u' = u_v, " = ", u_k - y_k & \text{for } u_\pm = '=', y_\pm = \\ u' = u_v, "+ = ", u_k + y_k & \text{for } u_\pm \in \{"+ = ", "- = "\}', y_\pm \in \{"+ = ", "- = "\}, 0 u_\pm u_k y_\pm \\ u' = u_v, "- = ", u_k + y_k & \text{for } u_\pm \in \{"+ = ", "- = "\}', y_\pm \in \{"+ = ", "- = "\}, 0 u_\pm u_k y_\pm \\ \emptyset & \text{for } u_\pm \in \{"+ = ", "- = "\}, y_\pm \in \{"+ = ", "- = "\}, 0 u_\pm u_k y_\pm \end{cases}$$

where $u_v = y_v$ For full updates we define $\forall v \in V U'_v = u_v \bigoplus y_v$

We refer to the preceding location before edge e as $\bullet e$ and the destination after taking the edge as $e\bullet$. Similarly, the sets of ingoing and outgoing edges from a location l is referred to as $\bullet l$ and $l\bullet$ respectively.

$\forall e = \{l_1, g, u, g', l_2\}$: $l_1 = \bullet e, l_2 = e\bullet, e \in \bullet l_2, e \in l_1\bullet$

A state s is defined as l, W where $l \in L$ is a location, and W is a function assigning values to all variables, such that $\forall v \in V : W(v) = \mathbb{Z}_{\geq 0}$ and $0 \leq W(v) \leq max$. We refer to S as the powerset of states.

A state satisfies a guard, $s \vdash g$, if $\forall v \in V : W(v) \vdash k$.

Invariants assign guards to locations.

$I \subseteq \{L \times G\}$

A transition $s, \xrightarrow{e}, s'$ is the result of taking an edge such that $s = \{l_1, W\}, e \in E = \{l_1, G_1, U, G_2, l_2\}, s' = \{l_2, W'\}$ where $\forall g \in G_1 : s \vdash g, \forall g' \in G_2 \cup I(l_2) : s' \vdash g'$, $\forall v \in V : W'(v) = u(W(v))$

A query ϕ is defined as a subset of states which we wish to know whether or not they are reachable: $\phi \subseteq S$, We use $l \in \phi$ as a shorthand for any states in the query that are in location l.

3.0.2 Reachability

In verification asking whether a state is reachable given an initial state is known as a reachability query. Given an initial state and model does a sequence of transitions exist $s_{init} \xrightarrow{e_1} s_1 \xrightarrow{e_{...}} s_{...} \xrightarrow{e_n} s_n | s_n \in \phi$ such that we can perform legal transitions from the start state to reach any state specified by the query.

3.0.3 Exploration Algorithm

This section defines the exploration part of the algorithm, for the purpose of establishing problem areas in partitioning and reductions.

As we concern ourselves primarily with reductions and partitioning, we use only a basic exploration algorithm with no optimizations. While this will skew performance testing in favor of reductions, we aim to test on the number of states explored, so exploration time is not a factor in evaluating the algorithm in this stage.

We first describe a conventional reachability algorithm for exploring the state space. This is done in algorithm 1

```
Data: Model M, initial state to explore from s_0, target states s_final
Result: boolean answer to φ
initialization;
waiting queue ← s_0;
while waiting queue ≠ ∅ do
    s ←pop(waiting queue);
    l ← location of s;
    for edges e originating from l do
        if e's guards are satisfied by s then
            newstate ← update effects on s;
            if newstate ∉ passed list then
                passed list ← newstate if newstate ∈ s_final then
                    return
                end
                true;
            end
        end
        waiting queue ← newstate;
        discard newstate and go to next edge;
    end
    go to next edge;
end
return false
```

Algorithm 1: Basic exploration algorithm - Keeps a queue for unexplored states and a list of all passed states. For every state in the waiting list we get all reachable states in one step. If a state is not in passed list it is added to the waiting queue, unless it is part of the query in which case we terminate with a positive answer. Once the waiting queue is empty and we have not found a positive answer we terminate with a negative one.

We next consider a basic algorithm for CBR in algorithm 2. We first need to redefine states as stateclusters that allow us to represent a group of states as $S_c \subseteq L \times G$. Now when we explore $s' \xrightarrow{e} s$ going backwards, so taking s as the original state and s' as the new state cluster after exploring edge e,

$\forall s* \in S | s* \xrightarrow{e} s : s'_c \xrightarrow{e} s \implies s* \vdash s'_c$

For example, if e has no guards and a single update x" = "5 our s'_c would have the guard $g(x) = \{min, max\}$

Data: Model M, initial state to explore from representing ϕ s_0, partition queue P, target states s_{final}
Result: boolean answer to ϕ
initialization;
partition waiting queue $\leftarrow s_{final}$;
while *partition waiting queue* $\not\emptyset$ **do**
  waiting queue $\leftarrow$ pop partition waiting queue;
  while *waiting queue* $\neq \emptyset$ **do**
    s $\leftarrow$ pop waiting queue;
    $l \leftarrow$ location of s;
    for *edges e originating from l* **do**
      if *e's guards are satisfied by s* **then**
        newstate $\leftarrow$ update effects on s;
        if *newstate* $\notin$ *passed list* **then**
          passed list $\leftarrow$ newstate **if** *newstate* $\in s_0$ **then**
            Return true
          end
          if *newstate belongs to current partition* **then**
            waiting queue $\leftarrow$ newstate
          else
            add newstate to appropriate partition. If the partition it belongs to is higher priority we return to that partition
          end
        end
        discard newstate and go to next edge;
      end
    end
  end
  go to next partition;
end
return false;

Algorithm 2: Basic CBR algorithm - The two differences are the addition of a queue for each partition and the reverse exploration. Although each partition has it's own waiting list, they all use the same passed list.

3.0.4 Partition Overreach

A new problem arises when reducing separate partitions. Consider the example in figure 3.2.

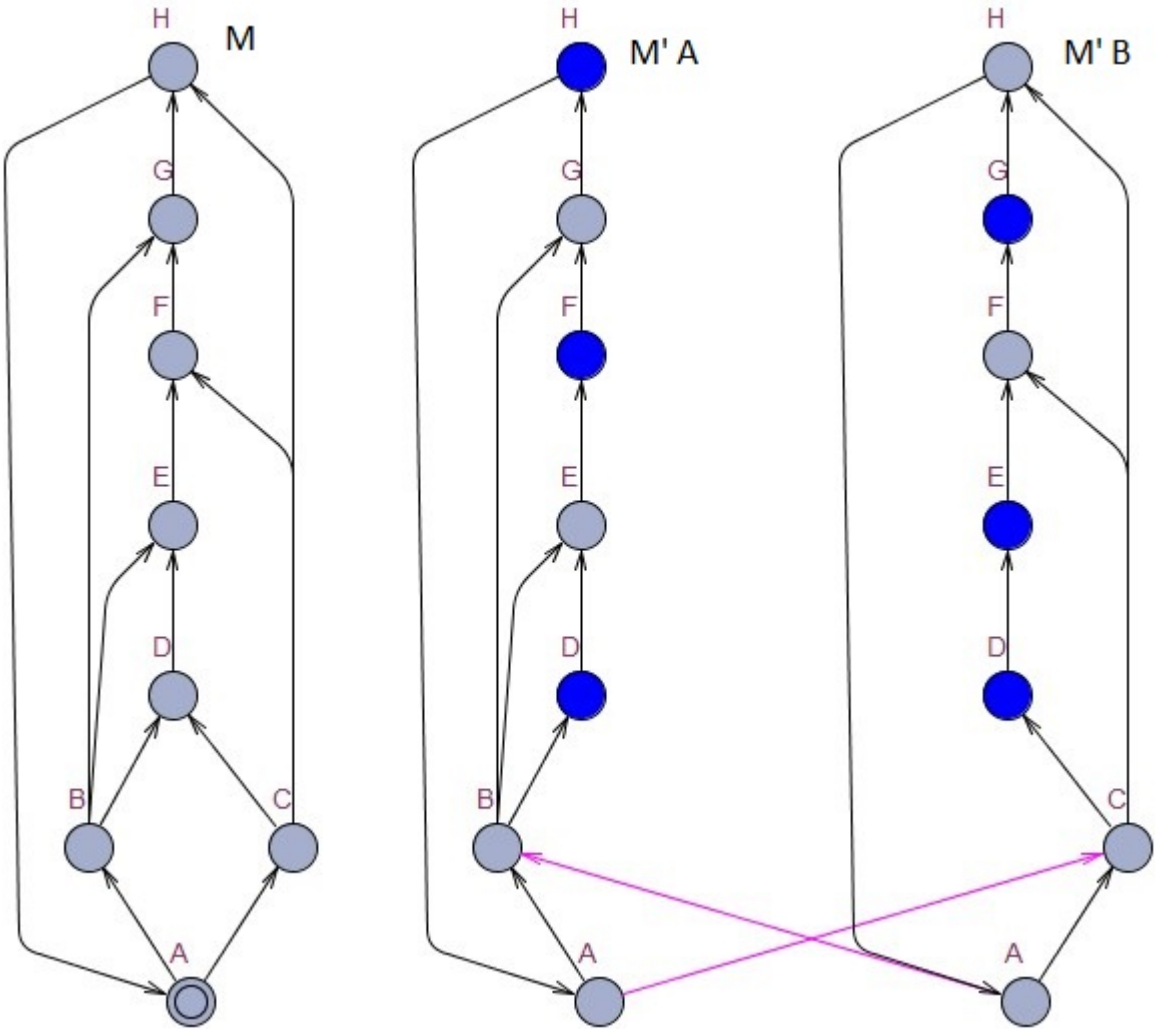

Figure 3.2: The Partition Overreach Problem. M is the initial model, $M'A$ and $M'B$ are partitions and blue locations are locations removed by the simple structural reduction introduced in the combination example. In the original model M if we have location E in our waiting queue and location F in our passed list, exploring E will yield no successor states. However, since F does not exist in partition $M'A$ if we explore E we will reach G.

The problem here is that the reductions reduce the state space within each partition, but since the removed states remain reachable in the other partition the reductions effectively do nothing. Since we're also removing intermediate states in this case duplicate elimination cannot occur until later, because the partitions have no common locations until returning to A.

The severity of this is at worst negating the gain from structural reductions performed after the partitioning

3.0.5 Problem Formulation

In the following chapters we answer the following:

- Given a simple integer automaton and a query; can structural reductions and partitioning be applied such that we maintain correctness and reduce the explored state space?
- Do partitions improve the performance of structural reductions?
- Can we preserve correctness across partitions?
- What can be done to avoid the Partition Overreach problem?
- How can reductions be applied efficiently?
- Where and how should partitioning be done?

Chapter 4

Algorithm Foundation

This chapter describes the overall structure of the algorithm, the exploration step and the architecture of the implementation.

With just structural reductions the algorithm has two primary steps:

- Apply Reductions
- Explore

We add the partitioning step after reductions, and repeat until we reach the condition for stopping partitioning, such as partition depth, described in algorithm 3.

4.1 Specification

As our goal is to examine the viability of a new algorithm we require a testbed that's highly configurable.

- Able to enable/disable specific reductions, ideally during execution
- Implementation for individual stages should not interfere with each other
- Performance of reduction step is important as it is repeated recursively, cannot reiterate over entire model after every reduction.
- Exploration performance not important, we will compare reduction performance by counting the number of states explored instead of time. As long as we do not explore redundant states testing should be accurate.

To keep the stages separate we use the visitor pattern for implementing Explorer, Reducer and Partitioner classes described in the following chapters, along with the support classes ReductionChecker and HeuristicChecker.

Data: Model M, initial state to explore from representing ϕ s_0, partition queue P, target states s_{final}
Result: boolean answer to ϕ
initialization;
$M \leftarrow Reduce(M)$;
partitions $\leftarrow M$;
while *partition depth ¡ desired depth* **do**
  $P \leftarrow$Pop(partitions);
  $M, M' \leftarrow$ partition(P);
  $M \leftarrow$Reduce(M);
  partitions $\leftarrow M$;
  $M \leftarrow$Reduce(M');
  partitions $\leftarrow M$;
  partition depth ++
end
while *partitions* $\neq \emptyset$ **do**
  P $\leftarrow$ Pop(partitions);
  if *BackwardsReachability(P)* **then**
    return true
  end
end
return false;

Algorithm 3: Partition-Reduce algorithm - Repeatedly partitions the model before exploration. Out-of partition states reached during exploration are added to their respective partitions.

4.2 Implementation of Model and Exploration Step

Based on algorithm 2 this includes the reduction and partition steps.

The implementation used in this project is described in the class diagram in figure 4.1

While this is only concerned with classes related exploration, reduction and partitioning steps are implemented using the visitor pattern, reusing the ModelVisitor interface.

Guard operations are handled in the Update and GuardCollections, implemented as dictionaries that map keys to values, specifically variables to guards. The GuardCollection returns a static member OPENGUARD if it contains no entry for a variable. OPENGUARD will accept any value between min and max. They return NULLGUARD the GuardCollection is unsatisfyable. This happens for example when you get the guard intersection between $x : 1 - 3$ and $x : 5 - 7$.

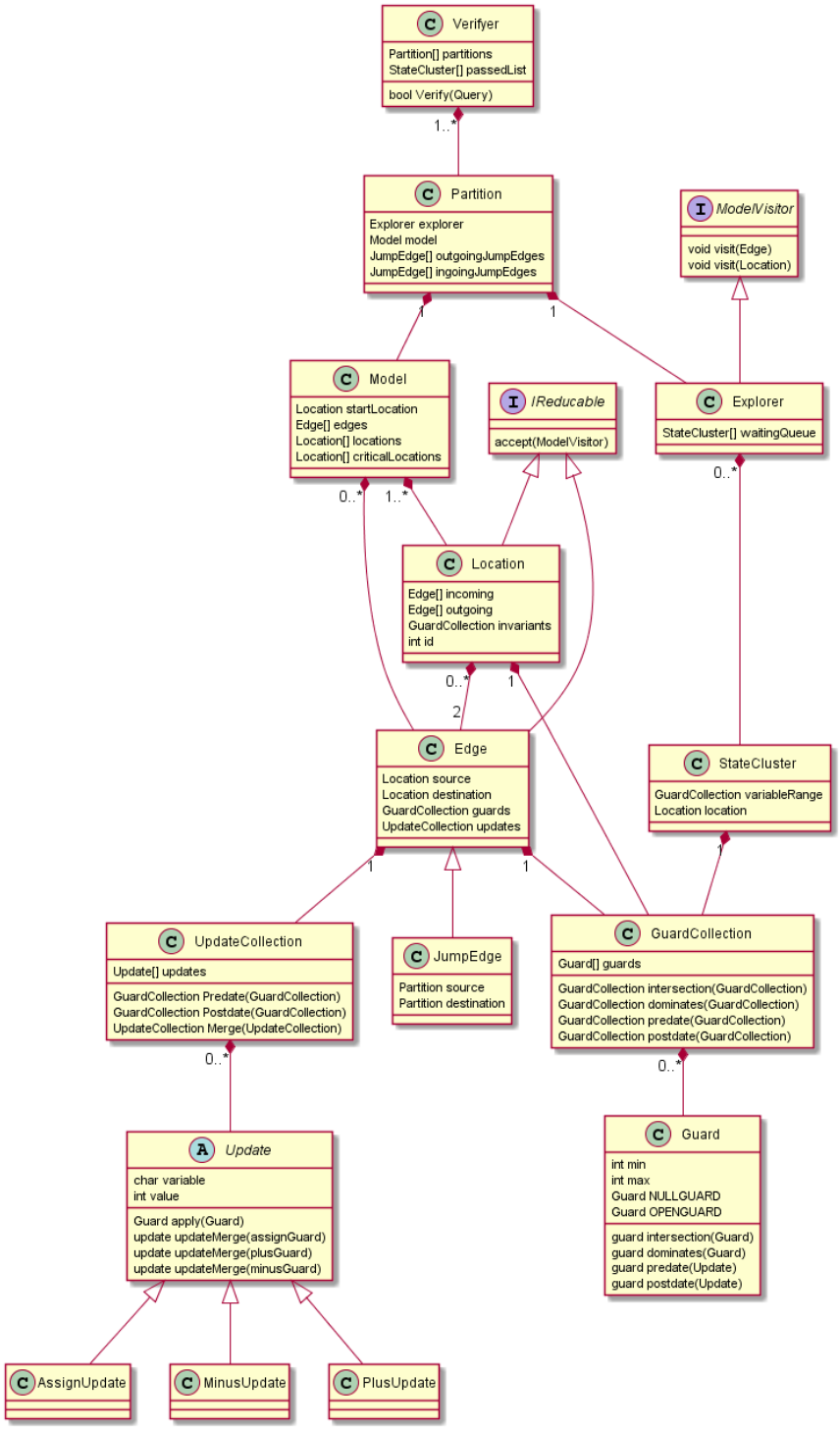

Figure 4.1: Class Diagram showing only classes relevant to exploration.

Chapter 5

Reductions

In this chapter we discus the reductions applied to each partition.

We define a reduction as follows.

A reduction is a function $R(M, \phi) = M'$ that when given a model M, and a query ϕ function is applied to M matching some condition, it produces a modified model M' such that $M \models \phi \iff M' \models \phi$.

5.1 Successor Boundedness

When applying reductions it is important that they do not alter the answer for the query. In theorem 5.1.1 we formulate the binary relation $\equiv_\phi$ between two models to mean that given the query ϕ the models will return the same result.

Theorem 5.1.1 *Models M and M' satisfy $M \equiv_\phi M'$ only if $M \models \phi \iff M' \models \phi$ where M, M' are models and ϕ is a reachability query.*

We next define a graph manipulation function as a function which given M, ϕ, l where M is a model, l is a location and $l \in loc(M)$ returns M' : $f(M, \phi) \longrightarrow M'$

In order for graph manipulation functions not to violate theorem 5.1.1 we introduce the attribute of successor boundedness in theorem **??**.

Theorem 5.1.2 *Successor Boundedness for a graph manipulation function, $F(M, l, \phi)$ implies that $M \equiv_\phi M'$ defined as $\forall s \in Succ^*(L_{init}) \exists s' \in Succ^*(L'_{init})$ such that $s = s'$ or $s' \in Succ^*(s)$ or $\nexists s_t \in Succ^*(s) | s_t \models \phi$*

The theorem can be read as for every state in the original model, there exists an equivalent state in the reduced model, or the original model's state can perform transitions to reach a state which is equivalent or it can never reach a state that would satisfy the query.

For graph manipulation functions that create modified models, this means that when removing parts of a model, the states removed have to either satisfy

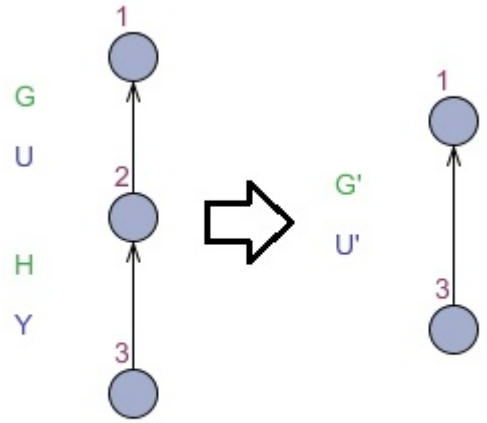

Figure 5.1: Path Reduction example. Location 2 is cut away and the two edges connecting it are merged into one.

$\nexists s_t \in Succ^*(s) | s_t \models \phi$ or $\exists s' \in Succ^*(s_t) | s' \in Succ^*(s)$, where s_t is a removed state from M' and s is a reachable state in M.

We show successor boundedness holding in a single step here:

$$\begin{array}{ccccc} s_0 & \xrightarrow{e_1} & s_1 & \xrightarrow{e_2} & s_2 \\ || & & & & || \\ s'_0 & & \xrightarrow{e'} & & s'_2 \end{array}$$

The reduced model removes the intermediate state s_1, but $s_2 \in Succ^*(s_0)$ and $s'_2 \in Succ^*(s'_0)$ still holds.

Since the property holds for any model, it will hold for repeatedly applying functions that satisfy the successor bounded property as defined in lemma 5.1.3.

Lemma 5.1.3 *M and M'' satisfy $M \equiv_\phi M'' \iff M \equiv_\phi M'$ and $M' \equiv_\phi M'' | F_1(M, l_1, \phi) = M'$ and $F_2(M', l_2, \phi) = M''$*

5.2 Path Reductions

A Path Reduction applies when we have a state with a single ingoing edge and a single outgoing edge. We merge both edges and their guards and updates into a new edge e':

Condition

A location l_1 with a single ingoing and outgoing edge. $\exists l$ such that $\bullet l = \{e_1\}$ and $l\bullet = \{e_2\}$ and $l \notin L^{crit}$

5.2.1 Function

We define $M' = \{L', L^{init}, L^{Crit}, E', V, min, max\}$ where $L' = L \setminus l, E' = E \setminus \{e_1, e_2\} \bigcup e'$ where $e' = \{l_1, G', U', l_2\} | l_1 = \bullet e_1, l_2 = e_2 \bullet$. We specify the guards and updates for e' as follows:

The new update function U' is the upmerge of e_1 and e_2: $U' = U(e_1) \bigoplus U(e_2)$

The guard is the Guard Intersection of $G(e_1)$, the guard intersection $G(e_2) \cap I(l)$ predated by $U(e_1)$ meaning any state taking the edge must satisfy the guard for e_1 and the guard for e_2 and invariant for l after applying the update from e_1. in $G' = G(e_1) \cap (G(e_2) \cap I(l)) \triangleleft_{U(e_1)}$

If the guard cannot be satisfied by any state which can happen if $G(e_1) \times G(e_2) \triangleleft_{U(e_1)}$ the function instead removes the location l and both connected edges: $M' = \{L', L^{init}, L^{Crit}, E', V, min, max\}$ where $L' = L \setminus l, E' = E \setminus \{e_1, e_2\}$.

5.2.2 Proof

The proof falls into two simple cases:

- Successor exists
- No Successor exists because we cannot satisfy ϕ

We show that theorem 5.1.2 holds for path reductions. To show this, we need to show that $\forall s \in l$ either $Succ^*(s) \not\models \phi$ or $\forall s_1 \in Pre(s) \exists s_2 | s_2 \in Succ^*(s_1)$.

We begin by examining the regular reduction function. As per our condition, l has only a single ingoing edge, thus $s_1 = Pre(s) \in \dot{l}$ If s satisfies the query, then there must be s_2 still reachable by $s_1 \xrightarrow{e'} s_2$ and our first case for theorem 5.1.2 holds.

If there is no transition $s \xrightarrow{e_2} s_2$, s can only satisfy the query if $s \models \phi$ but by our condition $l \notin L^{crit}$ that cannot be the case.

In the second case where the location is removed, again by $l \notin L^{crit} s$ cannot satisfy the query and successor boundedness is preserved.

5.2.3 Parallel Reduction

Parallel reductions take two edges going from and to the same locations and combines them into one as long as they have identical update functions.

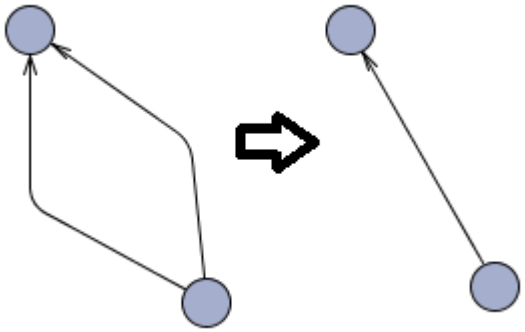

Figure 5.2: Parallel Reduction

Condition

The conditions for this reduction is that the two edges have the same locations as input and output, they must have identical update functions and have continuous guards:

- Edges $e1, e2 | \bullet e1 = \bullet e2, e1\bullet = e2\bullet$
- $U(e_1) = U(e_2)$
- $G(e_1) \not\asymp G(e_2)$

Function

We remove the matching edges and add a new edge with the same update and the guard intersection between the old edges:

$M' = M \setminus e1, e2 \bigcup e'$ where $e' = \{\bullet e1, G(e_1) \bigcup G(e_2), U(e1), e1\bullet\}$

Proof

Parallel reduction maintains all reachable states, as $s \xrightarrow{e'} s' \implies s \xrightarrow{e_1} s'$ or $s \xrightarrow{e_2} s'$

As condition two requires the edges have identical updates every state reachable via e_1 will be identical to e_2, and the guard union means that either $G(e_1)$ or $G(e_2)$ will be satisfied.

5.2.4 Dead End Reduction

A Dead End Reduction removes locations that have no outgoing edges which are not part of L_{Crit} as exploring them can never result in a positive answer.

Condition

The condition is a location with no outgoing edges that is not a part of our critical locations and thus cannot impact the query if removed:

$\exists l | l \notin L_{Crit}$ and $e\bullet = \emptyset$

5.2.5 Function

On a match we remove the location and all ingoing edges from the model.

$M' = M \setminus (l \bigcup \bullet l)$

Proof

Trivial as reaching l is not part of the querry and there is no way to progress from l: $\nexists e$ such that $s \xrightarrow{e} s' | l \in s$

Island Reduction

This reduction takes a location l with no ingoing edges, $\bullet l = \emptyset$, and removes the location along with all outgoing edges.

5.2.6 Guard Propagation

Guard Propagation does not normally remove locations, it just propagates guards through the graph and only removes if it creates an unsatisfyable guard.

If every outgoign edge has a guard, every ingoing edge gets has the predate of that guard and its update function added.

Condition

For a location there is a guard intersection for all outgoing edges

$\exists l | \forall (e_1, e_2) \in l : e_1 \setminus e_2 \neq$

5.2.7 Function

Modify ingoing edges to prevent exploring states that can't proceed one more step:

$\forall e \in l : G(e') = G(e) \setminus G' | G' = e_1 \setminus e_2$

Proof

Trivial as reaching l is not part of the querry and there is no way to progress from l: $\nexists e$ such that $s \xrightarrow{e} s' | l \in s$

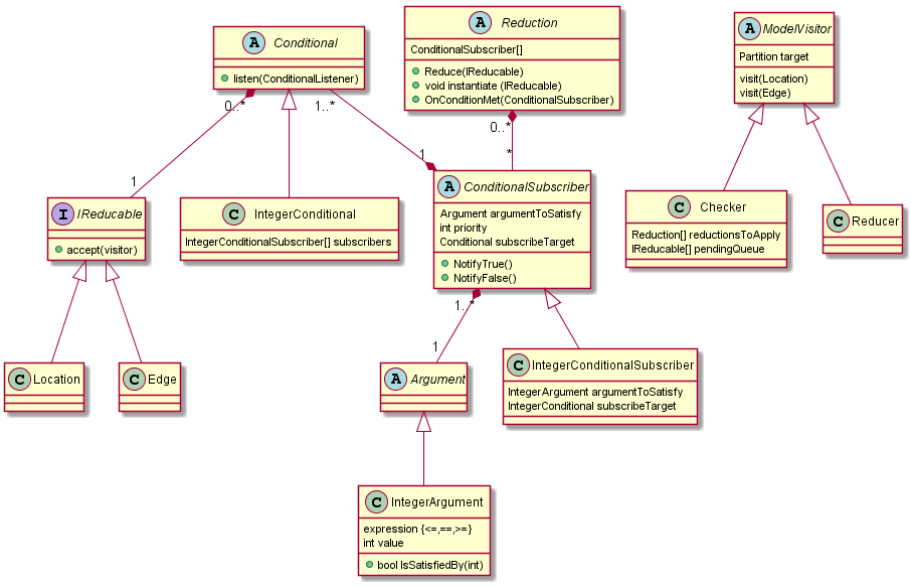

Figure 5.3: Class diagram for condition checking and reductions.

5.3 Reduction Implementation

We extend the implementation in 4.2 to add reductions. Our new components as described in the class diagram 6.1 are Conditionals, ConditionalSubscriber and Reduction, and the two visitors Checker and Reducer.

- Conditionals are instantiated on locations or edges and are responsible for checking the conditions specified by reductions. They use the observer pattern with IReducable as subject and ConditionalSubscriber as subscriber.
- ConditionalSubscriber subscribes to Conditionals with an argument and a priority.
- Reductions instantiate ConditionalSubscribers then wait for them to be satisfied. If a reduction takes multiple conditions it will instantiate the second after the first is satisfied and continue until a full chain subscribers are satisfied, then it will apply the Reduce method on the relevant subscriber target.

The reduction step is initiated by the Checker being called on the partition. The checker goes through all edges and locations, where it calls Instantiate for each reduction in it's list, as shown in the event diagram in figure 5.4. This creates the relevant conditionals at the current location if they are missing, and creates a subscriber for the reduction at the conditional.

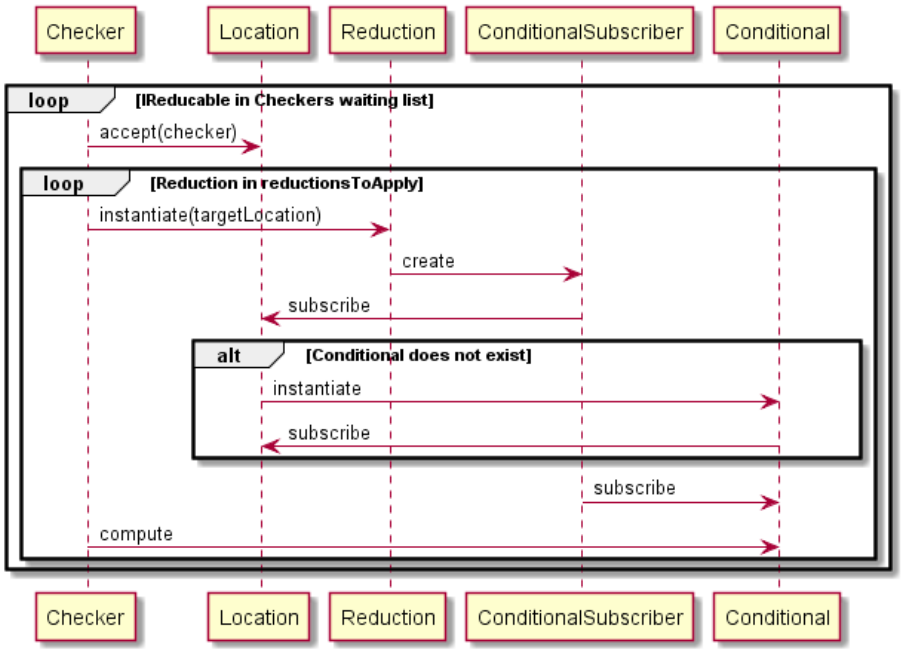

Figure 5.4: Event diagram for condition checking. This gives the example for path reductions where the reduction requires two Conditionals; IngoingCountConditional and OutgoingCountConditional to both have a value of 1.

Once created the conditional computes its result. The Checker will keep going through the model until all reductions have been instantiated on their locations.

Compute depends on the exact Conditional, for example IngoingCountConditional will get the number of ingoing edges at the location it is subscribed to and save the value as result. It will then go through all ConditionalSubscribers subscribed to it and see if result satisfies the argument they subscribed with.

The effect of the compute method can be seen in the other sequence diagram in figure 5.5. Once a ConditionalSubscriber has it's argument satisfied it will notify the Reduction that owns it that it has been satisfied. The reduction will then either require more subscribers to be satisfied, leading to a repeat of figure **??** without involving the checker. If the Reduction is fully satisfied it will apply it's function on the model, using the Reducer to remove model components. The reducer is responsible for cleaning up references to the reducable, as well as calling the Checker on adjacent reducables to re-calculate any conditionals that may have changed, for example removing an ingoing edge will cause IngoingEdgeConditional to recalculate.

The implementation of the reduction stage satisfies our stated goals as follows:

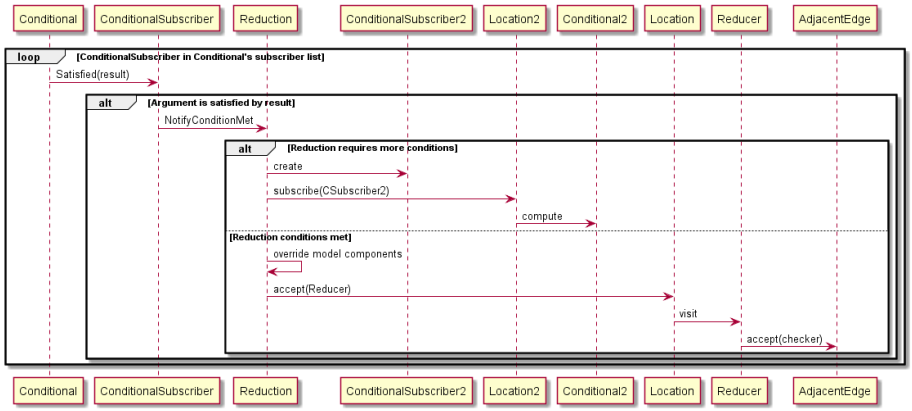

Figure 5.5: Event diagram for condition satisfied. This gives the example for when a ConditionalSubscriber is satisfied by a Conditional.

- We can enable and disable different reductions whenever a new partition is created. Each partition has a separate instance of Checker which contains the list of reductions to apply, so we can for example skip computationally heavy reductions after x partitions have been applied.
- The reduction step only scans the partition once during the initial pass by the Checker, and only the new partition. The old partition rechecks relevant conditionals via the Reducer as necessary.

Please note that while the conditional setup and use of visitors comes with a relatively high overhead for simple conditionals like the IngoingCount conditional which only has to get the number of ingoing edges in a location when it is computed. We justify this overhead on the basis that time spent on condition checking would primarily come from more complex conditionals involving guard comparisons where the overhead is negligible, the overhead does not increase time complexity, and the conditional structure is reused during Heuristic calculations as described in section 6.5.

Chapter 6

Partitioning

In this section we discus how we partition the model. First we need to define partitions and partitioning:

6.1 Partition Definition

A Partition P is a Simple Integer Automaton with a new set of edges, E_P which is the set of edges connecting partitions together. They are defined as $l_1, P', l_2 | l_2 \in P'$. For our partition P either $l_1 \in P$ or $P' = P$. As with regular edges l_1 is the origin, l_2 is the destination, but l_2 does not exist in the same partition as l_1. P' refers to the partition of the destination.

Partitioning is a graph manipulation function similar to structural reductions with a condition and function as well, but unlike reductions a partitioning returns two partitions.

For Partitioning we require a stricter condition that preserves all states as defined in theorem **??**.

Theorem 6.1.1 *A graph manipulation function* $F(M, l, \phi) \longrightarrow M' | l \in M'$ *is state preserving if* $\forall s \in l(M), \forall s' \in l(M') : Succ^*(s) = Succ^*(s')$

Basically state preservation just means that Partitionings do not alter the state space. Since any function that maintains all successor functions automatically satisfies the successor boundedness theorem 5.1.2 we have lemma 6.1

Theorem 6.1.1 $\implies$ theorem 5.1.2

We next wish to show that we can eliminate duplicate states when encountered in seperate partitions, as specified by theorem 6.1.2

Theorem 6.1.2 *If* s *is reachable in* P, s *is reachable in* P' *and* $F(P, l, \phi) \longrightarrow P'$ *satisfies theorem5.1.2 then* $\exists s_1 \in P, Succ^*(s) | s' \vdash \phi \iff {}_2 \in P', Succ^*(s)'$

We first consider partitioning without reductions. Since partitioning preserves all states and transitions, the conditions holds trivially as exploring s in any partition will yield all the same successor states.

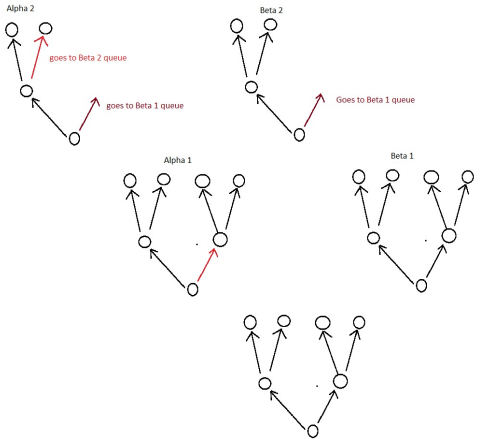

Figure 6.1: By repeatedly applying partitioning functions we break the model into many smaller models.

Next we consider what happens when we apply reductions. by theorem 5.1.2 reductions must maintain at least one successor state that either satisfies ϕ or has a successor state which does. Thus if $s \vdash \phi$ in partition P then $s \vdash \phi$ in partition P' as well, so it is sufficient to explore either one.

6.2 Split Partition

We've been referring to partitioning via split partitions so far. A split partition starts at a location with multiple outgoing edges where it selects an edge. It copies the model into a new partition, then converts all outgoing edges to jump edges leading into their respective locations in the other partition.

This means the other edges are removed as ingoing edges from their destinations in the new partition, which can enable reductions such as unreachable reductions or path reductions. It also removes the edge used to enter the new partition from its destination in the original partition.

6.3 Partition Overreach

There is another interesting property for partitioning, which is not only can we skip states that have been explored in other partitions, but we can also skip states that would be reachable, but have had their location removed by reductions and have no successors. These states can never satisfy the query, as defined in theorem 6.3.1.

Theorem 6.3.1 *If s is reachable in P and $Succ(s) \in P = \emptyset$ then $forall P' s \nvdash \phi$*

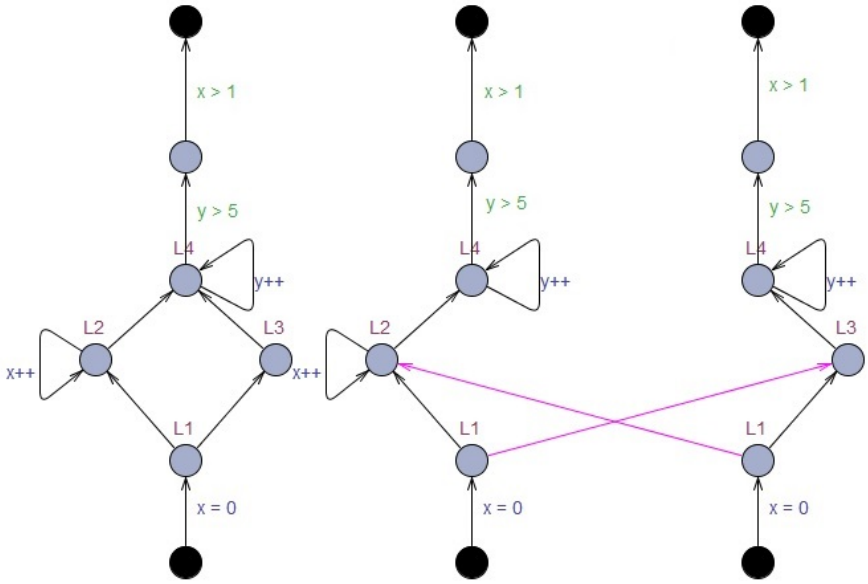

Figure 6.2: Partition Overreach example. We partition using split partition on $L1$. Note that any state has to use the loop on $L2$ to satisfy the guard leaving the section.

We prove this by contradiction. Suppose that $existss'$ such that $s' \in Succ*(s)|s' \vdash \phi$ then whichever function removed the location was not successor bounded.

We examine this in figures 6.2 and 6.3

What is interesting here is that if we explore the right partition first, we end up eliminating duplicate states in $L4$ for the left partition; $\{L4 : x = 0, y = 0\}, \{L4 : x = 0, y = 1\}, ...$ although these states have reachable successors in the left partition, none of them can satisfy the $x > 1$ guard,but some can still reach $L5$.

This shows us two things:

- Exploration order can be used to mitigate partition overreach.
- Applying reductions, such as removing $L4$ from the right partition can make partition overreach worse.

While the second point is technically true, reductions often lead to more reductions and should overall always be worth it. One way to take advantage of this could be to mark reduced locations that contain a predecessor of a lot of

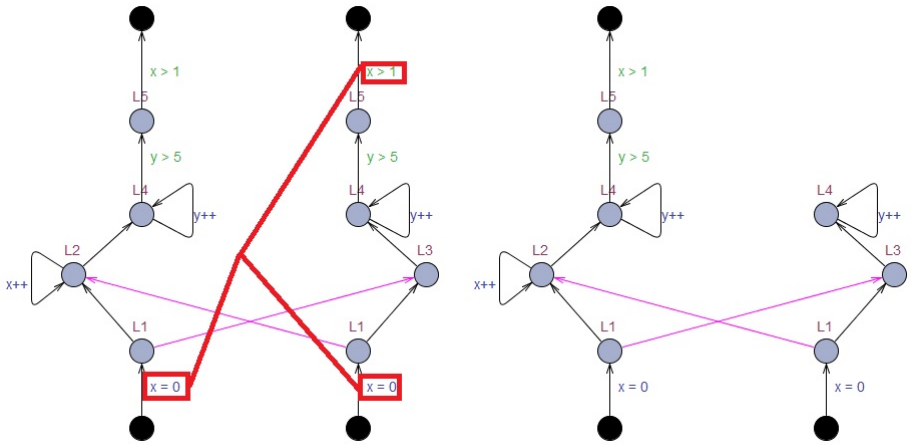

Figure 6.3: Partition Overreach example. Since the right partition can never satisfy the guard leaving $L5$, we can remove that edge without violating successor boundedness, although it is not a defined reduction. We can then apply a deadend reduction on $L5$ leaving the right partition stuck in $L4$

successor states across partitions, give it priority during exploration and deny reductions on this location in other partitions, but that is beyond the scope of this report.

We introduce a new edge type for edges leaving the current partition. These edges have an origin and a destination as usual, however it is not required that the origin exists in the α partition, only in the β partition (Relevant for reductions). In addition they refer to a specific queue as partitioning is nested. When these edges are taken, the state cluster performing the action and the edge are placed in the β queue if unique.

When a partition has no states remaining in its pending queue, we move onto the next partition until there are no remaining partitions to explore.

6.4 Guard Partition

Given that the cause of partition overreach is shared successor states, a partitioning that more resembles the CBR partitioning by splitting the state space between partitions would help negate it. For this we introduce the Guard Partitioning.

A guard partitioning is done on a single edge, which is split into two with mutually exclusive guards. Consider the model in figure 6.4

We create a guard split on the red edge splitting around $x = 5$ which gives us two partitions in which all ingoing jump edges have either $x > 5$ or $x <= 5$.

We can use this to determine that the $x > 5$ guards in the right partition of figure 6.5can never be satisfied,we remove those edges and repeatedly apply

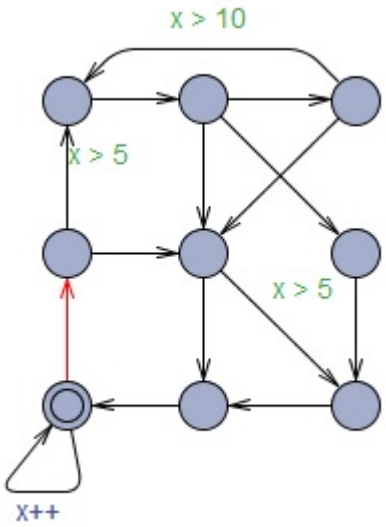

Figure 6.4: Guard Partition example. Note the multiple guards with $x > 5$. if we create a guard partition around this so that only one partition can satisfy those guards it will allow us to remove those edges.

deadend and unreachable reductions to get figure 6.6.

Unlike edge partitioning, guard partitioning is capable of making no progress for reducing a model as we can create many partitions with small guards. We would need a good heuristic for locating edges and guards to perform guard splits.

6.5 Partitioning Implementation

In this section we describe the third stage of the algorithm, in which we find a suitable location then split the model on that location into two partitions.

As seen in the class diagram in figure 6.7 we reuse the model components from the reduction phase, specifically Conditionals and their subscribers. Subscribers have a new value added, which represents the heuristic value of meeting the argument of that subscriber.

Our heuristic is calculated by getting the conditionals to check which subscribers would become satisfied if it's result changed to some other value, and then pass it the value we expect it to take after applying the partitioning. For example, Split Partitioning on some edge would reduce the ingoing count for the following location by 1, so we pass -1 as argument to GetBestHeuristic on the IngoingCountConditional and that will give us a value indicating that some reduction will become available if we perform the split there.

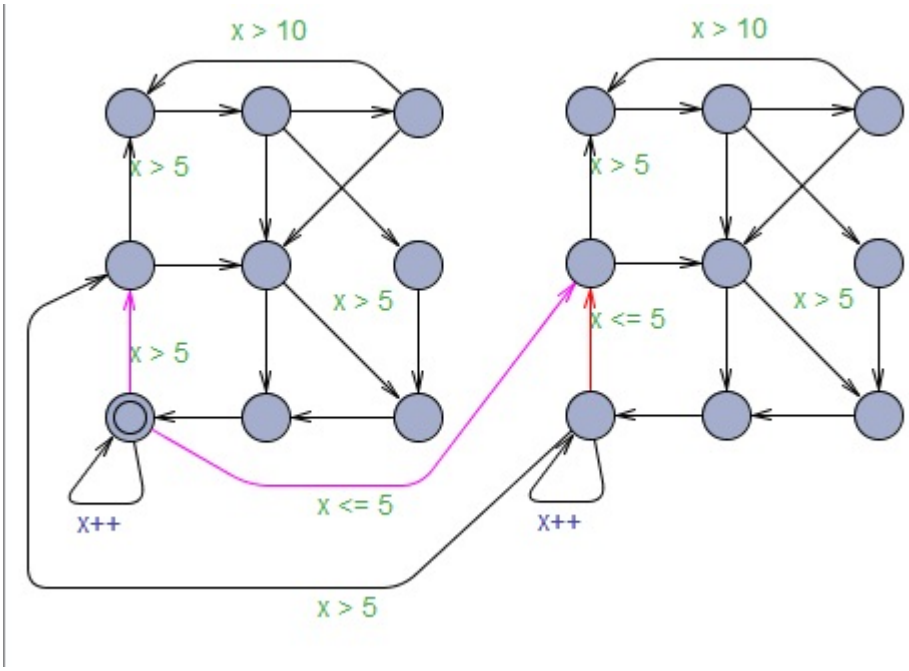

Figure 6.5: Model after guard split. All new jump edges have added guards to them.

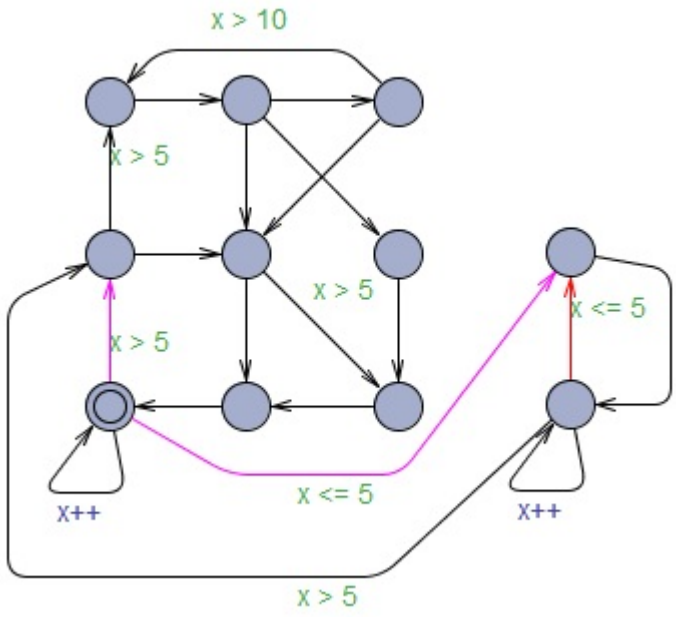

Figure 6.6: Reductions after guard split

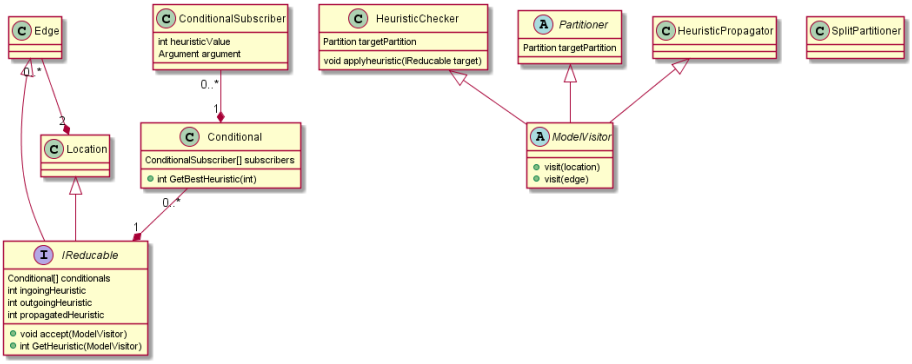

Figure 6.7: Class diagram for relevant classes during partitioning phase.

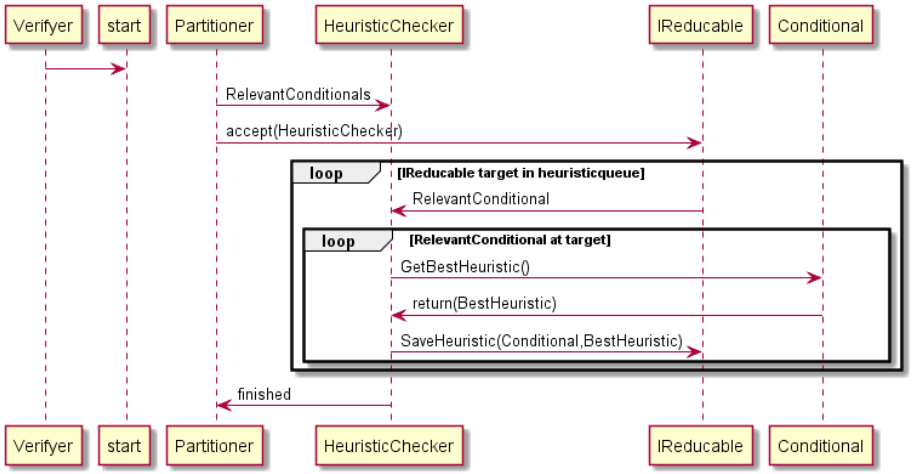

Figure 6.8: Sequence Diagram for finding the base heuristic for a Conditional type

This process is described in the Sequence Diagram for Heuristic Checker in figure 6.8. Here the Partitioner gives the HeuristicChecker a list of the conditionals that are relevant for it. In the case of split partitioning that's ingoing and outgoing count. Next it starts the HeuristicChecker on the reducables where it gets the value for each relevant Conditional and saves it on the reducable. Note that this means if nothing has subscribed to the Conditional during the reduction phase nothing is calculated, so if path, deadend and unreachable reductions are disabled, we don't need to do anything else the HeuristicChecker will assign no values.

After the HeuristicChecker has assigned values on all reducables for each conditional the HeuristicPropagator is started. This visitor is for propagating the values across the model, for example by adding the ingoingCountHeuristic to all outgoingCountHeuristic on the successors of the current reducable. This is described in figure 6.9.

The Propagator keeps track of the highest encountered heuristic value for each partitioner, and once it has finished propagating it starts the appropriate partitioner on the reducable with the highest value.

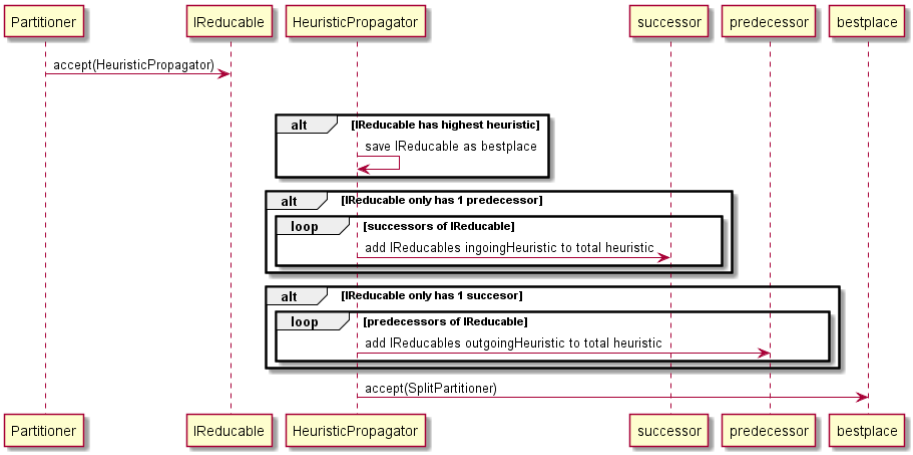

Figure 6.9: Sequence Diagram for propagating heuristic values

Chapter 7

Discussion

In this chapter we discus the results of the paper. We have introduced structural reductions with simple variable semantics where they can be shown to work well especially with backwards reachability. We examined the viability of combining CBR with structural reductions and have formulated an experimental algorithm.

7.1 Algorithm

We have introduced a new algorithm for combining reductions with partitioning and proven correctness of the algorithm and basic reductions, however no performance testing has been done. We have shown the algorithm to be viable and have not found any reason for it to increase time complexity from O(Model Size) with reductions to beyond $O(2\times$ Model Size), as reductions, heuristic search and partitioning traverse the model in its entirety only once, with additional checks only when reductions are applied. Since the application of reduction guarantees great performance payoff for constant time only the condition search is considered relevant. We have highlighted a key problem area with combining structural reductions and partitioning, Partition Overreach, which can lose the advantage of structural reductions in the worst case scenario. While we believe this to be unlikely, testing is needed to determine the actual impact. We also show that the problem can be mitigated to some extent by manipulating the exploration order to eliminate states that have no successors able to answer the query earlier.

The implementation can be found at https://github.com/misterpaws/Partition-Reduce/releases/tag/0.9 (Refers to implementation at time of hand-in)

7.1.1 Testing

As no implementation is complete, this section discusses the tests that would need to be performed in order to determine the usefulness of the algorithm.

While we have proofs for the correctness of reductions, as the paper shows

there are some new challenges when combined with partitioning. Testing the algorithm against an existing model checker such as UPPAAL, as well as testing results between unpartitioned and partitioned models would help showing we have not missed anything.

Testing the time of reducing and partitioning a model compared to the time it takes to explore is a key component as we can essentially keep partitioning forever, but at some point it needs to stop. While reductions take much less time than exploration without partitioning, the more we break the model down the smaller the state space the reduction is actually applied to so there are diminishing returns when applying partitiong recursively.

Testing the severity of partition overreach both in general and worst cases should also be done.

Testing heuristics for priority of reductions and partitionings is a key concern as partitionings do not ensure any improvement at all, and thus carefully selecting them to optimize potential reductions will improve the algorithms performance.

With the partitioning creating copies of states and edges, even if we do not increase the state space, the memory cost of all the new models may become a concern and should also be tested for.

7.1.2 Future Work

Creating an implementation for testing is necessary to determine how well it actually performs, as the paper has only concerned itself with correctness of the algorithm.

Further exploring the partition overreach solution in section6.3

The most crucial if the algorithm is to see any use will be expanding the semantics for parallel automata and potentially timing. While timing will place more constraints on structural reductions, parallel semantics should not be a big problem.

Several types of structural reductions exist that can be applied to the model in addition to the four discussed in this paper. Adding more reductions and partitionings will improve performance of the algorithm by increasing the state space reduction.

Developing good heuristics for which reductions and especially partitionings to apply when is also important.

Depending on test results looking into ways to handle successor boundedness could also be important.

Bibliography

[1] Gerd Behrmann et al. "Developing UPPAAL over 15 years". In: *Softw., Pract. Exper.* 41 (Feb. 2011), pp. 133–142. DOI: 10.1002/spe.1006.

[2] Frederik M. Bønneland et al. "Stubborn versus structural reductions for Petri nets". In: *J. Log. Algebraic Methods Program.* 102 (2019), pp. 46–63. DOI: 10.1016/j.jlamp.2018.09.002. URL: https://doi.org/10.1016/j.jlamp.2018.09.002.

[3] Kim Guldstrand Larsen, Florian Lorber, and Brian Nielsen. "20 Years of Real Real Time Model Validation". In: *Formal Methods*. Ed. by Klaus Havelund et al. Cham: Springer International Publishing, 2018, pp. 22–36. ISBN: 978-3-319-95582-7.

[4] Jørn Lind-Nielsen et al. "Verification of Large State/Event Systems Using Compositionality and Dependency Analysis". In: *Formal Methods Syst. Des.* 18.1 (2001), pp. 5–23. DOI: 10.1023/A:1008736219484. URL: https://doi.org/10.1023/A:1008736219484.