Master's Thesis Software Science

Radboud University Nijmegen

Should I wait for supervisory controller synthesis to finish or change the settings?

Author:
Dion Bremer
s1020299

First supervisor/assessor:
dr. ir. Dennis Hendriks

Second assessor:
prof. dr. Frits Vaandrager

May 21, 2026

Abstract

When performing supervisory controller synthesis using the ESCET toolkit, a user has no indication of how long the algorithm will take before it terminates. In fact, a user does not know whether they should wait until it finishes, or if they should try again using different settings. In this master thesis, we explore the symbolic computations used in the synthesis algorithm to determine when a set of settings produces a bad run of the algorithm. In particular, we explore the saturation algorithm used for performing symbolic reachability computations. We consider the algorithm from three perspectives: the input of the algorithm, its inner workings and the implementation of the algorithm. We find that when using bad settings, the algorithm performs a lot of duplicate work, which can be measured during a run. In the end, we come up with a predictor using this data on duplicate work that can detect a bad run when having seen 20% of that run, with 82% accuracy.

Acknowledgments

Foremost, I would like to thank my thesis supervisor Dennis Hendriks for his continuous support. His help, insights and our weekly meetings were very helpful to completing this thesis project.

Next, I would like to thank TNO-ESI[1] for allowing me to conduct my master thesis at their organization, and allowing me to get a taste of the world of applied research. In particular, I would like to express my gratitude to the Poka Yoke project team, Dennis Hendriks, Wytse Oortwijn, Andrea Peruffo, Jos Hegge and Dennis Arets, and fellow students Calvin Terpstra and Terence Beijloos, who allowed me to be a part of the group, and who were always available for answering questions or chatting about the current state of the research.

Lastly I would like to thank Frits Vaandrager, who took the time and effort to be the second examiner for this master's thesis.

[1]See `https://esi.tno.nl/`.

Contents

1 Introduction **4**

2 Background information **6**
- 2.1 Supervisory controller synthesis using the CIF tooling . . . 6
- 2.2 Symbolic EFAs . . . 6
- 2.3 Fully-reduced ordered BDDs . . . 7
- 2.4 The synthesis algorithm . . . 8
- 2.5 The saturation algorithm . . . 10
- 2.6 The JavaBDD library . . . 11

3 Related work **13**

4 Distinguishing good and bad synthesis runs **15**
- 4.1 Approach . . . 15
- 4.2 Benchmark models . . . 16
- 4.3 Input perspective: the evolution of the states BDD . . . 17
- 4.4 Inner workings perspective: the application of transition relations . . . 19
- 4.5 Data structure perspective: implementation in JavaBDD . . . 27
- 4.6 Evaluation . . . 40

5 Slowdown predictors **42**
- 5.1 Node duplicates versus cache entry duplicates . . . 42
- 5.2 Slowdown predictors . . . 42
- 5.3 Data collection . . . 42
- 5.4 Predictor tuning . . . 46
- 5.5 Predictor validation . . . 53

6 Conclusion **57**

7 Future work **58**

1 Introduction

In the current day and age, a world without cyber-physical systems is unthinkable. They include, but are not limited to, medical imaging machines, modern cars, factory equipment and waterway locks. These cyber-physical systems are composed of two parts: a physical one and a virtual one. The physical part interacts with the world around it. For instance, sensors in a conveyor belt detect where products are moving, induction loops in front of traffic lights detect waiting cars, and medical scanners detect whether a patient is in the machine. These systems then use this data to control a series of actuators, in order to respond to certain events. For example, when the induction loop detects that there is traffic on one side of the road but not on the other, it decides to turn the light to green, and let the traffic through safely.

This communication between sensors and actuators is facilitated by the "cyber" part: software that determines which actuators can be activated, under which circumstances. Engineering such software is anything but trivial. Aside from functional requirements, there are many safety requirements that the system must at all times adhere to, in an environment where unexpected events may occur at any time. For example, when a robot in a factory moves between stations, it must stop immediately if it suddenly detects a worker in front of it.

To combat this complexity, the software consists of a hierarchy of supervisory controllers. At the lowest level of the hierarchy, a resource controller controls the operation of a single sensor or actuator. One level higher, a supervisory controller controls several of these resource controllers, facilitating communication and ensuring some of the safety requirements are met for this part of the system. Another level higher, another supervisory controller controls several of these lower-level supervisory controllers, again allowing these to communicate and operate according to the (safety) requirements.

Even though this hierarchy of supervisory controllers offers a conceptual framework for the software, designing and implementing this hierarchy is still a complex task using traditional software engineering methods. However, over the years, more and more aspects of software engineering have become automated, or at least computer-aided. One of the steps in this automation is the introduction of formally defined models. These models capture the behavior and requirements of the system, allowing for automated techniques such as model checking and model-based testing for finding bugs, and even allow code generation from the model. Going one step beyond this automation, synthesis-based engineering allows a system engineer to synthesize models automatically, allowing them to focus on what a system should do, instead of how it should do it. The process is as follows: the engineer designs plant models, representing what (components of) the system can do, and requirement models, which represent what a system must or must not do. After this, the engineer synthesizes a correct-by-construction controller model at the click of a button. The engineer then needs to validate whether the original requirements were correct, using simulation. When the engineer deems the model to be correct, they can either use it to implement the controller in code manually, or generate code automatically.

One of the toolkits available for synthesis-based engineering is the Eclipse Supervisory Controller Engineering Toolkit, or Eclipse ESCET™[2] for short [1]. In order to synthesize a model using the ESCET toolkit, the engineer has to model the system's plant and requirement models as extended finite automata, in the CIF modeling language. In CIF, a plant model is a model that specifies which behavior can occur. For example, a plant model for a button would specify that the button can be pressed or not. A requirement model specifies which behavior must not occur. A CIF model may consist of several of these automata. Together, these automata represent a *state space*. It is well-known that the number of states in such a state space blows up exponentially. In order to be able to do efficient computations on this state space, the CIF model is converted into a symbolic representation, using *binary decision diagrams* (BDDs) [2]. The model is then synthesized, by performing reachability computations on these BDDs.

Even though this symbolic representation is generally efficient for synthesis, there are still models that are infeasible to synthesize [3]. There is also no indication on the progress that the algorithm makes, so that a user does not know how much more time the algorithm will take. This due to the fact that the synthesis algorithm is a *fixpoint* computation. The algorithm continuously trims states from the state space, until a safe, controlled system is left. Since it is not known beforehand how many states need to be trimmed, it is impossible to give an exact percentage on the progress that the algorithm has made.

An observation on the efficiency of the synthesis algorithm is that the settings that the user chooses can have a massive influence on the efficiency of the algorithm. For example, it is known that the variable order used for creating the BDDs can make an exponential difference on the size of the BDD [4], and therefore on the efficiency of operations on this BDD. Different settings can lead to slower or faster runs of the synthesis algorithm, even on the same model. We can compare runs of the same model with each other, using different settings. If a run outperforms most of the other runs, we call it a *good* run. Otherwise, it is a *bad* run. Since the synthesis algorithm represents state spaces as BDDs and performs computations on these BDD, the BDDs ultimately determine whether a run will be good or bad. Therefore, inspecting the behavior of the BDDs during the synthesis algorithm could provide an insight into whether a run is good or bad. If a run is indeed bad, we

[2]See also `https://eclipse.dev/escet/`. 'Eclipse', 'Eclipse ESCET' and 'ESCET' are trademarks of Eclipse Foundation, Inc.

would like to communicate this to the user as early as possible. The user could then act on this, by terminating the run, and choosing different settings. This brings us to the research question: is it possible to determine, having seen a small fraction of a run of the synthesis algorithm, whether the run is good or bad?

In order to answer this question, we investigate the BDDs that are at the core of the synthesis algorithm. These BDDs are used to perform state-space explorations. To do this, CIF uses the state-of-the-art *saturation* algorithm [5] for its reachability computations. The saturation algorithm itself is also a fixed point computation. It takes a BDD representing certain states of the system, and repeatedly applies transition relations to this BDD. This creates a new, intermediate BDD. It then applies transition relations to that BDD, again creating a new BDD. Once the newly created BDD is the same as the previous BDD, the algorithm terminates. The efficiency of this algorithm thus depends on how often transitions are applied, and on which BDDs these are applied. Also, since the same set of transitions can be applied many times, it is not unthinkable that some BDD operations may be performed multiple times. These duplicate operations could also be detrimental to a run.

This thesis aims to answer the research question by studying the saturation algorithm from the three aforementioned perspectives: the evolution of the intermediate BDDs, the application of transition relations, and the occurrence of duplicate computations. Since this is an exploratory study, we employ a structured approach to study each of the three perspectives, using different experiments. From these results, we determine which of these three directions is the most promising. Then, we construct a heuristic using the insights gained from this perspective, that determines whether a run of the synthesis algorithm is good or bad. The final step is to determine whether this heuristic can also predict whether the run is good or bad, after having seen a small part of a run.

From studying the three perspectives, we find that the duplicate computation perspective is the most promising for creating a heuristic. More precisely, the creation of duplicate nodes is used as an indicator of how good or bad a run is. Using this data, several predictors are created and validated. This predictor is then tuned using more experimental data. Lastly, this predictor is evaluated, using separate validation data. It is shown that it can predict that a run is bad with 82% accuracy, while having only seen 20% of the run.

The rest of this thesis is structured as follows. Section 2 contains background information on the synthesis algorithm and the symbolic algorithms used in synthesis. Next, Section 3 describes work related to progress made in symbolic algorithms, and the efficiency of these algorithms. Section 4 details the structured approach in finding an indicator for synthesis progress. In the next section, Section 5, we further investigate the duplicate computations performed by the algorithm, and we tune and evaluate a predictor for synthesis progress. Section 6 contains a conclusion and discussion, and lastly Section 7 contains suggestions for future work.

2 Background information

This section contains relevant background information on the CIF tooling, the synthesis algorithm and symbolic computations.

2.1 Supervisory controller synthesis using the CIF tooling

The main tool for performing supervisory controller synthesis in the ESCET toolkit is the CIF tooling[3]. It supports the complete development process for synthesizing correct-by-construction controllers for *discrete-event systems*.

A single CIF model can consist of multiple *plant* and *requirement* models. Plant models describe what a system can do, such as the physical behavior of a system. A plant model is therefore a model of the uncontrolled system, and the two terms can be used interchangeably. Requirement models on the other hand aim to limit this uncontrolled behavior, and model what a system must or must not do.

The modeling of plants and requirements is based on *extended finite automata* (EFAs) [3]. An EFA consists of *locations*, connected with directed *edges*. It is always in one location at a time, its *current* location. Each edge is labeled with an *event* of the discrete-event system. When an event occurs, the EFA takes the edge corresponding to that event, thus ending up in the next location. Note that self-loops are allowed, so this next location is not necessarily different from the original. When a system is modelled with many EFAs, then the combination of all the current locations of the individual EFAs corresponds with the state of the system. The automata are further extended with *variables*, *guards* and *updates*. An EFA can define variables, that can be updated by the edges. A guard is a boolean predicate over some of the variables, that can be attached to an edge. That edge may then only be taken if the guard evaluates to `true`. This allows for precise specification of the system.

After the system has been modeled, the plant and requirement models are combined in parallel. Here, they *synchronize* on shared events. This means that when an event occurs, all EFAs must take an edge corresponding to that event. If an event is missing from the local state of a plant model, it is considered physically impossible. On the other hand, if it is missing from the states of a requirement model, it is considered undesirable behavior, and must be restricted by the controller. This way, the synthesis algorithm ensures *safety*.

Events can be *controllable*, or *uncontrollable* [3]. When an event is controllable, then the controller determines when it occurs or not. For example, an event where a robotic arm is moved is controllable. An uncontrollable event is not controlled by the controller. For example, an event where an induction loop detects a waiting car, is uncontrollable, as the controller has no control over whether a car waits on the sensor or not. Controllable events can be prevented by the controller, but uncontrollable events cannot. If an uncontrollable event must be prevented, the synthesis algorithm does so by preventing controllable events leading to that uncontrollable event. This way, the algorithm ensures *controllability*.

Safety and controllability together are sufficient for creating safe systems. However, consider once again the example of the traffic light with the induction loop. The simplest way of creating a safe, controllable supervisor for this system is to simply always leave the traffic light on red. In order to have some more control over the behavior of the system, the user can *mark* states of the model. The resulting supervisor makes sure that at least one marked state can always be reached. This way, the resulting controller is *non-blocking*. The synthesis algorithm guarantees that the properties of safety, controllability and non-blockingness are obtained in a *maximally permissive* manner, meaning that the system is minimally restricted in order to obtain these properties.

As mentioned earlier, the separate EFAs are composed by synchronizing them on events. This produces a *linearized* model. This model contains a single location with self-loops. It still represents the same behavior, but in a single EFA. Here, the locations of all EFAs are represented using variables. For example, if there are two EFAs in the model, each with 3 possible locations, then the linearized model would contain two variables l_1 and l_2, whose possible values are 0, 1 and 2. Transitions of the EFAs are then represented by guards on self-loop edges, using these location variables. The advantage of creating such a linearized model is that the entire model behavior is captured by the edges of the linearized model. This allows for the model to be described by predicates.

2.2 Symbolic EFAs

It is well known that synchronizing EFAs can cause the number of possible states of the model to blow up. This is a problem for the reachability computations on which the synthesis algorithm is based, as searching the state space one state at a time becomes infeasible. To combat this state-space explosion, the synthesis algorithm uses a symbolic representation of the system, where many states can be represented and explored at once. This

[3]See also `https://eclipse.dev/escet/cif/`.

symbolic representation is known as a *symbolic extended finite automaton* (SEFA). Since this is a broad topic, the information given in this section is limited, and we refer the interested reader to [3].

Using the linearized model, the uncontrolled system can be represented symbolically using boolean predicates. These predicates represent sets of states. The way this works is as follows. Consider a predicate $P : \{0,1\}^n \to \{0,1\}$ over n variables. This predicate can now be interpreted as the characteristic function of a set S, where each element is a satisfying assignment of the predicate. When each vector of variables is interpreted as a state, then the set S represents a set of states, as in Equation 1.

$$S = \{(v_0, \ldots, v_n) \in \{0,1\}^n | P(v_0, \ldots, v_n) = 1\} \tag{1}$$

Aside from representing states, we can also represent the edges of the model using predicates. We will refer to an edge represented by a predicate as a *transition relation*. In order to construct this representation, the boolean variables are divided into *old-state* variables and *new-state* variables. Given an old-state variable x, we denote its new-state variable by x^+ [3], doubling the number of boolean variables. We make sure that the states described earlier consist only of old-state variables. The new-state variables are only used in transition relations.

By combining old-state and new-state variables using boolean functions, we can construct predicates that represent transition relations. A satisfying assignment for this predicate can be divided into its old-state and new-state variables. The old-state variables indicate the state of the model that the transition relation can be applied to, and the new-state variables dictate the result of applying the transition. For example, applying the transition $x \wedge \neg x^+$ on state x results in $\neg x^+$, which then becomes the next current state $\neg x$.

The operation as described above is known as a `relnext` operation [6]. This operation is defined as $relnext(s,t,v) = \exists_v (s \wedge t)[V^+ := V]$. Here, s is a set of states, t is the transition relation, and v is the set of relevant variables. This set v is necessary to be able to apply partial transition relations. If a variable is not in v, then it is implied to remain unchanged by the transition. First, `relnext` computes $s \wedge t$. Since we want to compute the next state, we need to consider the new-state variables. Therefore, we remove the old-state variables using an existential quantification, resulting in a set of new-state variables. This quantification will result in the correct set, as long as there is a satisfying assignment using the old-state variables. Lastly, the term $[V^+ := V]$ indicates that the new-state variables are replaced by their corresponding old-state variables, since new-state variables are only used in transition relations.

There are a few variations on this operation. First, there is a `relprev` operation. This operation works similarly to `relnext`, but instead interprets the given set of states as the result of applying a transition relation. It then computes the previous set of states, essentially taking an edge in the opposite direction. Both `relnext` and `relprev` have 2 additional variants, namely a union and intersection variant. `relnextUnion` takes an additional set of states u as parameter, and computes $relnextUnion(s,t,v,u) = relnext(s,t,v) \vee u$, in one optimized operation. `relnextIntersection` takes an additional set of states i and computes $relnextIntersection(s,t,v,i) = relnext(s,t,v) \wedge i$, in an optimized operation. The variants for `relprev` behave similarly.

Formally, a SEFA is a 6-tuple $(V, D, \Sigma, E, p_0, p_m)$ [3]. Here, V is its finite set of variables, D contains a finite domain of possible values for each variable, Σ is its *alphabet*, that is, its finite set of events, E is its finite set of edges, p_0 is the predicate representing its initial states and p_m the predicate representing its marked states.

2.3 Fully-reduced ordered BDDs

The symbolic computations in the synthesis toolkit are performed using *binary decision diagrams* (BDDs), as implemented in the *JavaBDD* library[4]. These BDDs represent boolean predicates as rooted, directed, acyclic graphs [2], and allow for efficient manipulation of these predicates. There are several variations of BDDs, such as *zero-suppressed BDDs* [7], *tagged BDDs* [8] and *RexBDDs* [9], but the ESCET toolkit uses *fully-reduced ordered BDDs* [2], which we will refer to simply as BDDs.

BDDs are graphs that consist of *decision nodes* and *terminal nodes*. There are two terminal nodes, which represent the boolean values `true` and `false`. These terminal nodes are often called *one* and *zero* respectively. The decision nodes are associated with boolean variables. As such, these nodes have two outgoing edges, called the *high* and *low* edges. The high edge corresponds to when the associated variable evaluates to `true`, and the low edge similarly corresponds to `false`. See Figure 1 for an example of two BDDs, both representing the same boolean function. Here, a solid arrow represents a high edge, and a dotted arrow represents a solid edge. When evaluating a BDD, simply start at the root, evaluate the corresponding variable, and take the edge corresponding to that evaluation, which leads to a different node. Then, evaluate the variable corresponding to this node. Repeat this process until a terminal node is reached. This terminal node then represents the outcome of the function.

The variables in an ordered BDD are *totally ordered*. This means that for variables x and y with $x < y$, nodes corresponding to variable x are closer to the root than nodes corresponding to variable y. As a consequence,

[4]See `https://github.com/com-github-javabdd/com.github.javabdd`.

the nodes of the BDD are divided into *levels*, where each level corresponds to a certain variable. The root of the BDD has the lowest level. However, when showing a BDD visually, we often draw the root at the top of the BDD, which is counterintuitive. Therefore, we use the term *layer* for discussing traveling up and down a BDD, where lower layers are closer to the terminals, and higher layers are closer to the root. The levels follow the opposite direction: a lower level is closer to the root, and a higher level is closer to the terminal. The same boolean function has many different representations, obtained by changing the order of the variables. In the synthesis tool, once an order for the variables is selected, it is not changed anymore. Also, when operations are performed on multiple BDDs, then all of these BDDs need to adhere to the same variable order. The set of variables that appear in a BDD is called the *support* of the BDD.

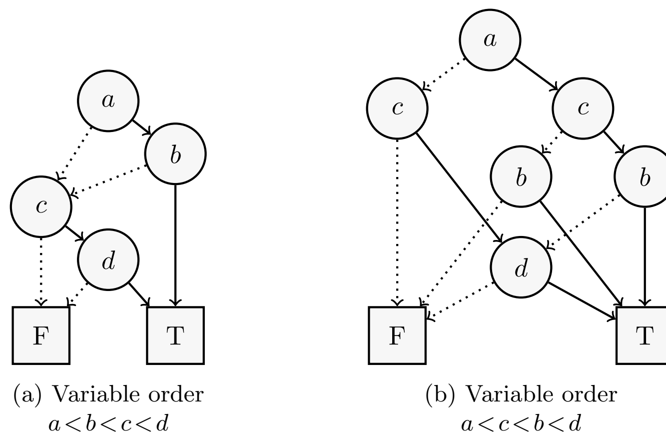
In order to keep the BDD representation compact, every node in a BDD is unique. Two nodes are the same when their levels are equal, their low edges lead to the same node, and their high edges lead to the same node. If this is the case, the node can be represented just once, where it has multiple parent nodes. Additionally, the BDDs in the synthesis tool are *fully-reduced*, for an even more compact representation. This means that the BDD does not contain *don't-care* nodes, where the low and high edges lead to the same node. It is well-known that the uniqueness of nodes, the fully-reducedness and the total order together lead to a canonical representation for boolean functions [2].

The variable order can have a massive impact on the performance of BDD operations, as choosing a different order could make an exponential difference in the number of nodes needed to represent the same boolean function. Figure 1 shows a BDD representing $(a \wedge b) \vee (c \wedge d)$, using two different variable orders. Note the difference in the number of nodes between these BDDs. Finding an order that can represent the function with the fewest amount of nodes is NP-complete [10]. For that reason, the synthesis tool implements several heuristics to try and find a good order. These heuristics aim to find the minimal *total span*, or the minimal *weighted event span* [11]. The total span metric is calculated by taking the sum of the spans of each event, where the span is defined as the difference between the top and bottom variable. The weighted event span is a weighted variant on this, where variables higher in the BDD carry more weight. The synthesis tool also has several algorithms that attempt to optimize these metrics. Some of these operate on the variable order globally, while others attempt to make local updates to this order. On a global level, the *FORCE* algorithm [12] attempts to find a good variable order by having variables that occur together in events gravitate to each other in the order. Like the FORCE algorithm, the *DCSH* algorithm [13] also attempts to place variables that appear together often close to each other. Contrary to FORCE, however, it uses *dependency structure matrices*, and reduces the bandwidth of these matrices. Lastly, the *sliding window* algorithm [4] attempts to locally optimize the order, by making permutations of subsets of the variable order, and selecting the best one. Aside from these algorithms, the synthesis tool supports performing these algorithms sequentially, reversing orders, selecting the best order out of multiple and choosing different initial variable orders.

2.4 The synthesis algorithm

When the model is represented symbolically using an uncontrolled system (or plant) SEFA, the synthesis algorithm can be applied to compute the controlled system. The synthesis algorithm in CIF is based on the algorithm by [14]. It is a *fixpoint* computation. By performing several reachability computations, it continuously strengthens guard predicates on edges until a fixed point is reached, and forbidden states (by the requirements) have become unreachable in the controlled system.

Algorithm 1 provides the pseudocode for the synthesis algorithm. This algorithm takes the uncontrolled



(a) Variable order $a < b < c < d$

(b) Variable order $a < c < b < d$

Figure 1: Two BDDs representing $(a \wedge b) \vee (c \wedge d)$ for different variable orders. The order on the left results in a BDD with only 4 nodes, while the order on the right results in 6 nodes. Image taken from [3].

Algorithm 1 Symbolic Supervisory Controller Synthesis (*SSCS*), taken from [3]

Require: Plant SEFA $P = (V, D, \Sigma, E, p_0, p_m)$ and forbidden states p_f over V.
Ensure: Controlled-system SEFA S.

1: $C \leftarrow \neg p_f$
2: **repeat**
3: $\quad C' \leftarrow C$
4: $\quad C \leftarrow BRS(p_m, E, C)$
5: $\quad B \leftarrow BRS(\neg C, E_u, true)$
6: $\quad C \leftarrow \neg B$
7: $\quad C \leftarrow FRS(p_0, E, C)$ ▷ Optional step.
8: **until** $C = C'$
9: **for all** $e \in E$ with $\sigma \in \Sigma_c$ **do**
10: $\quad g \leftarrow g \wedge relprev(e, C)$
11: $S \leftarrow (V, D, \Sigma, E, p_0 \wedge C, p_m \wedge C)$

system, in the form of the uncontrolled SEFA, and a set of forbidden states, and computes the controlled system. This controlled system is represented by a set of states that are safe, controllable, nonblocking and maximally permissive. The algorithm starts by initializing the set of controlled states C as he complement of the set of forbidden states in line 1. The goal of the algorithm is to restrict C until this set is safe, controllable and nonblocking. It does this in a fixed point computation, that starts at line 2. At the beginning of this fixed point computation, a copy is made of C, which is later used to decide whether the computation is done. The algorithm then repeatedly computes sets of nonblocking, controllable and reachable states[5], until the resulting set of states does not change anymore. First, at line 4, the set of nonblocking states of C is calculated by performing a backward reachability search, starting from the marked predicate p_m, and bounding this result to be a subset of C. It then updates C to be equal to this new set of nonblocking states. Then, the algorithm computes the set of bad states for this new C. Since C represents the set of good states, its complement represents the set of bad states. The algorithm then computes more bad states, by performing a backward reachability search using the edges labeled with uncontrollable events, in line 5. This pushes back the bad states through these uncontrollable events, revealing more bad states, which are stored in the set B. Now, in line 6 we can find the set of good states again by complementing B. Optionally, on line 7, a forward reachability operation can be performed, to determine which states are actually reachable by the system. This step is optional, as the plant already encodes which states are reachable. It can make the result of the synthesis algorithm more intuitive for the user, however, as it would no longer represent states that the controlled system cannot reach. Also, it can have a big impact on the performance of the algorithm, as some models synthesize faster with forward reachability, and others slower. In line 8, it is checked whether the fixpoint operation has completed, by determining whether the operations have made an impact. If the set of states C is different from the copy C' that the algorithm made earlier, then we need to do the reachability computations again, as the algorithm has trimmed states from the state space, revealing new potentially bad states. If no change is detected, the algorithm strengthens the system guards g for controllable events in Σ_c in lines 9-10. This way it makes sure that the system stays within the computed controlled states C. Lastly, in line 11, the initial and marked states are restricted to be within the controlled states C, and the resulting controlled system SEFA is returned.

Algorithm 2 Backward reachability search (BRS), adapted from [3] to match the current version of the synthesis tool.

Require: Starting predicate S, edges E, restriction R
Ensure: The coreachable states S', that is, those states that can reach states in S via edges in E, restricted to states in R.

$E \leftarrow E.sort()$
$V \leftarrow \phi$
for all $e \in E$ **do**
$\quad V \leftarrow V \cup support(e)$
if $R = true$ **then**
$\quad P' \leftarrow saturationBackward(S, E, V, i, 0)$ ▷ See Algorithm 3.
else
$\quad P' \leftarrow boundedSaturationBackward(S, R, E, V, i, 0)$ ▷ See notes under Algorithm 3.

[5] The order of these computations can be set by the user prior to performing the synthesis.

2.5 The saturation algorithm

When a symbolic representation for the states and transition relations is chosen, one can use several strategies to perform the reachability computations. There are multiple possibilities, but the preferred strategy in the synthesis tool is the *saturation* strategy [5]. It is a state-of-the-art algorithm that, empirically, outperforms other reachability algorithms. This section provides a concise treatment of the algorithm. For a more detailed overview, we refer the reader to [5] and [15].

To perform saturation, the transition relations created by the synthesis tool are put in a list and sorted, so that the levels of the root nodes are sorted low to high. The transition relations are also implicitly divided into groups. Transition relations belong to the same group when their top nodes have the same level. Now, the saturation algorithm traverses both the states BDD as well as the list of transition relations. It moves up and down through the structure of the states BDD, as well as left and right in the list of transition relations. It first traverses all the way down the states BDD, and all the way right in the list of transition relations. It then applies this last group of transition relations, whose root nodes are closest to the terminal nodes out of all transition relations. This produces a small intermediate BDD, as it is close to the terminals. The same group of transitions is then again applied to this BDD. This process continues until the BDD does not change anymore, so that this is a fixpoint computation. At this point, we say the BDD node is *saturated*. Then, the algorithm moves back up in the states BDD, and to the left in the transition relations. It then applies this next group of transition relations to its current states BDD. If the states BDD changes, the algorithm saturates the bottom-right again, before attempting to saturate using the current transition relations. This way, the algorithm moves back and forth, up and down, until finally the root of the states BDD is saturated. Intuitively, the algorithm is efficient due to this up and down strategy. When a sub-BDD has been saturated, then the nodes above it can be saturated more easily, given that the already saturated nodes do not change much anymore. This is of course especially true when every group of transition relations acts on different variables. This way, the size of the intermediate BDDs remains close to the size of the final result.

Algorithm 3 shows pseudocode for the saturation algorithm, based on its implementation in the JavaBDD library. In fact, it shows the pseudocode for the *backward saturation* algorithm. Here, the backward in the name reflects the fact that it is a backward reachability computation. Saturation takes as input the BDD representing the set of starting states s, a list of transition relations t, a list of sets of variables v, corresponding to the variables used in the transition relations, and an index c representing the current transition relation to apply. Additionally in JavaBDD, the algorithm has an *instance number* parameter i, which is used to store and look up results from previously applied saturation operations in an operation cache. The list of transition relations, and thus also the variable sets, needs to be sorted based on the top variable, as mentioned earlier. Furthermore, the algorithm uses a few helper functions, namely: `LEVEL`, `ISONE`, `ISZERO` and `makeNode`. `LEVEL` returns the level of a node, `ISONE` returns whether a node is the terminal node `one`, and `ISZERO` returns whether a node is the terminal node `zero`. The `makeNode` function creates a new BDD node at a given level, with given low and high child nodes.

The algorithm starts by checking whether a base case is reached in lines 2 and 3. Line 2 tests whether the given states BDD is a terminal node. If so, no transition relations can be applied to this node, and it is returned instead. Line 3 checks whether the algorithm has already applied all transition relations, by checking whether the current transition relation is out of bounds. If so, it again can simply return the current states. Next, in lines 4 and 5, the operation cache is consulted. This cache stores BDD operations that it has already performed. If the result of the current operation is present in this cache, it can return the previously computed result.

If the current parameters do not constitute a base case and no result is found in the cache, the algorithm has to actually perform the computation. Here, there are two cases. The first case is that the states node is above that of the set of variables for the current transition relation c. Here, nothing has to be done on this level. Instead, as seen in lines 7-9, the saturation algorithm travels downward: the algorithm is performed recursively on the low and high children of the states node, and a node is constructed using the results. In the second case, when the states node is at equal level, or below the current set of variables, the actual saturation takes place. First, in line 11-13, the current group of transition relations is determined. These are determined based on having the same top level. The variable n marks the end of this group, so that the current transition relations to apply are those with indexes in the range $[c..n)$. In line 14, we store a copy of the states BDD in a variable r. This variable represents the result of the operation, as well as the intermediate BDDs. Next, in lines 15-20, the fixpoint computation is performed in a loop. First, in line 16, the algorithm moves to the right in the list of transition relations, and saturates the intermediate result using those transitions. Then, before saturating with the current group of transition relations, the intermediate result is stored in line 17. This intermediate result is useful for determining whether or not the fixpoint computation has completed. In lines 18-19, the current group of transition relations is applied to the intermediate result, producing new intermediate results. The `relprevUnion` operation makes sure that new states are added to the intermediate result, without removing states. When the transition relations have been applied, the algorithm checks whether the fixpoint computation has been completed in line 20. If so, the algorithm breaks out of the loop, as the result of the computation is

now known. If not, the algorithm loops back to line 15, where the computation is performed again, but now on a different intermediate result. Finally, in lines 21-25, the operation cache is updated, and the result is returned in line 26. This result is the saturated node obtained by saturating s. When s is the BDD representing the starting states, this result is the set of backward reachable states from this set of starting states.

Algorithm 3 Backward Saturation

Require: Starting states s, list of transitions t, list of variables v, instance number i and current index c.
Ensure: Return the states node saturated from the current transition relation onward.

```
1: procedure SATURATIONBACKWARD(s, t, v, i, c)
2:     if ISONE(s) ∨ ISZERO(s) then return s
3:     if c = relations.length then return s
4:     cacheEntry ← cacheLookup(s, i, c)
5:     if cacheEntry.s = s ∧ cacheEntry.i = i ∧ cacheEntry.c = c ∧ cacheEntry.o = saturationBackward then
       return cacheEntry.result
6:     if LEVEL(s) + 1 < LEVEL(v[c]) then
7:         low ← saturationBackward(LOW(s), t, v, i, c)
8:         high ← saturationBackward(HIGH(s), t, v, i, c)
9:         r ← makeNode(LEVEL(s), low, high)
10:    else
11:        n ← c
12:        while n < v.length ∧ LEVEL(v[n]) = LEVEL(v[c]) do
13:            n ← n + 1
14:        r ← s
15:        repeat
16:            r ← saturationBackward(r, t, v, i, n)
17:            p ← r
18:            for all j ∈ [c..n) do
19:                r ← relprevUnion(t[j], r, r, v[j])
20:        until r = p
21:    cacheEntry.s ← s
22:    cacheEntry.i ← i
23:    cacheEntry.c ← c
24:    cacheEntry.o ← saturationBackward
25:    cacheEntry.result ← r
26:    return r
```

Aside from backward saturation, JavaBDD also contains `forward saturation`, `bounded backward saturation` and `bounded forward saturation`. In forward saturation, the set of forward reachable states is computed, by using `relnext` operations instead of `relprev` operations. In the bounded variations, the algorithm receives an additional set of states as parameter, that represents a restriction on this result. The result of such a bounded saturation operation is always a subset of this restriction. As seen before, the synthesis algorithm uses this restriction to make sure that the controlled system remains within the set of safe states.

2.6 The JavaBDD library

As mentioned previously, the ESCET toolkit uses JavaBDD, a single-threaded library for performing BDD operations. This library makes it possible to create and manipulate BDDs, using the factory programming pattern. The central object here is the `JFactory` object, as it is the object that stores and manipulates BDD nodes.

Under the hood, `JFactory` uses a *node table* to keep track of BDD nodes. This node table is sometimes also referred to as the *unique table*, as BDD nodes are unique. An entry in this node table consists of 5 integer values, that together represent a BDD node and some bookkeeping. The first integer stores the level of the node, as well as a reference count and a mark. The second integer is the address of the low child of the node. Here, address is simply the index of the node in the node table. The third integer is the address of the high child of the node. The fourth integer is the hash value of the node. This hash is calculated over the level of the node and the references to its low and high children. Lastly, the fifth integer represents the next free entry in the node table. When creating a factory, the user specifies the initial size of this table, that is, the maximum number of nodes that it can store.

When more and more nodes are created, the table eventually fills up, and no new nodes can be inserted. When this happens, the factory attempts to free space in the table by performing a garbage collection. This

garbage collection happens using a mark-and-sweep algorithm, as follows. Before collection, the root nodes of the BDDs are given a reference count. From these roots, the rest of the nodes of these BDDs are recursively marked, to indicate that they cannot be collected. The terminals are not marked, but also never collected. Now, the factory goes through the table, removing nodes that are both unmarked and have no reference count. These include BDDs that have been freed manually by the user, as well as nodes that have become redundant due to BDD reduction rules. If garbage collection resulted in many free spots for nodes, operation continues as usual. However, if the number of free spaces is still below some threshold, the table can be resized. This resizing is different for small and large tables. If the size of the table is below some threshold, the size of the table is doubled. Otherwise, it is increased by a constant factor. This way, the table can arrange itself to find a good size. This is often preferred over using a large initial table size, since choosing a large size results in more memory usage, and can lead to worse performance due to CPU cache and page misses.

Aside from a node table, the `JFactory` object uses *cache tables* to keep track of which operations were performed already, and what the result of that operation was. It is known that these tables are essential to the performance of BDD operations [16]. This can be seen by the fact that a single BDD node can have multiple parent nodes. This means that when an operation traverses a BDD, it can come across the same node multiple times. If no cache table is used, that same computation has to be performed multiple times. By storing the result in a cache, the computation is only performed a single time. In fact, JavaBDD uses several cache tables, for different operations. The aforementioned relational operations and saturation operations are stored in the same cache table, however.

Like the node table, the cache tables have a finite size. JavaBDD allows for the size of these tables to be different from the size of the node table. This is important, as in practice there are often many more cache entries than BDD nodes. This also means that there is not enough space in the tables to store all previous computation results. Therefore, JavaBDD allows new cache entries to overwrite existing cache entries. The cache tables are also cleaned up, during the garbage collection of the node table. After the node table is cleared of unused nodes, the algorithm clears cache entries that still contain references to these removed nodes. The cache tables are also doubled in size, whenever the node table is resized, so that a growing number of nodes also corresponds to more available cache storage.

3 Related work

Since this thesis studies the progress that the synthesis algorithm makes, and in particular the progress that the reachability computations make, there are quite a few works that are tangentially related to this thesis.

Since we are interested in the progress that the synthesis algorithm makes, we should consider works that also address this topic. Unfortunately, there is not much work on this. There is work, however, on the progress that SAT-solvers make. SAT solving is a technique that is related to BDDs, where a satisfying assignment for a boolean predicate is found by searching through possible assignments of the boolean variables. There have been several attempts at estimating the progress that a SAT solver makes. This could be interesting, as SAT solvers can also be used to perform reachability computations [17]. Notably, the work by [18] seems most related to our work. Here, they use machine learning techniques to estimate the size of the search tree that needs to be searched until a satisfying assignment will be found. This is not directly applicable to our work, however, as our search space is in general not a tree. Also, the type of SAT solving considered is not a fixed point operation, so that it does not apply directly to our work.

If we look at BDD-based reachability computations only, then the issue of a computation stagnating is explicitly addressed in [19]. This stagnation is what we are interested in, as it could be the moment to communicate to the user that they should change the settings of the algorithm. The authors define *stagnation* to occur when too few new states are reached at each iteration of the fixed-point computations. They then attempt to overcome this issue by applying a new strategy to performing *high-density* fixed-point computations. Unfortunately, they do not come up with a detection mechanism of when a computation stagnates, however, and instead attempt to solve the issue directly. For this reason, this work is also not directly applicable to our work.

Since it seems difficult to find work related to the progress of the synthesis algorithm, we can instead turn our attention to works that attempt to estimate the efficiency of the algorithm. In the past, there have been attempts to predict the efficiency of symbolic supervisory synthesis based on the input of the algorithm. In [20], the authors use so-called *process communication graphs* to quantify the effort needed to perform symbolic reachability computations. In this way, they try to capture the intrinsic complexity of the model in a metric. They then use the insights gathered here to construct a reachability algorithm, called the *workset* algorithm, that computes a set of reachable states using a fixed-point computation. However, the authors do not come up with any progress indication for their algorithm. Also, their research focuses on the intrinsic complexity of the models, while we focus on accidental complexity, due to the settings used, regardless of intrinsic complexity. For these reasons, their work is not directly applicable to the research in this thesis.

Aside from looking at the efficiency of the synthesis algorithm, one can look more generally at predicting the efficiency of BDD operations based on the input of the operations. The number of operations needed to perform an operation on BDDs is related to the size of the input BDDs. Therefore, a smaller BDD representation leads to more efficient operations. The size of BDDs is influenced by the chosen variable ordering. In Section 2.3 we mentioned several heuristics for choosing a variable order, as described in [12], [4] and [13]. Related to this is the work by [21], where the complexity of BDDs is estimated using a mathematical model. This allows them to predict the size of a BDD for a given variable ordering, without explicitly building the BDD. However, this does also does not quite solve our problem. In fact, the saturation algorithm can start with a small BDD, but increase exponentially in size during a run of the algorithm. Aside from that, it is not known how the size of the input BDD relates to the number of fixed-point iterations needed. Therefore, this work is also not directly applicable to our research.

Another method of achieving smaller BDD representations is by using different reduction rules. Different types of BDDs can be more or less efficient than others for a particular problem. As mentioned earlier, the synthesis tool uses fully-reduced ordered BDDs [2], which has as a reduction rule that *don't care* nodes are not explicitly represented. Another type of BDD is the *zero-suppressed BDD* (ZDD) [7]. Like fully-reduced BDDs, every node in a ZDD is unique. However, the reduction rule is replaced by the rule that no high edge should point to the zero terminal node. *Tagged BDDs* [8] attempt to combine these two, by applying both reduction rules to create an even more compact representation of boolean functions. Edges in tagged BDDs are given a tag, that specifies up to which level a given reduction rule is used. Finally, *RexBDDs* [9] attempt to overcome the flaws of tagged BDDs by using nine reduction rules, which are stored on the edges between nodes. All of these different BDDs could influence the efficiency of the synthesis algorithm. However, even when using these different representations, the progress that the algorithm makes is still not known. It seems unlikely that a different representation completely solves the problem, and that using these representations the synthesis algorithm can still slow down considerably given the wrong set of settings. Therefore, we do not focus on different BDD representations in this thesis.

If we broaden the scope to more general algorithm runtime prediction, the work in [22] uses machine learning techniques to predict the runtimes for algorithms concerning SAT solving, mixed integer programming and the traveling salesperson problem. Using a large set of parameters for each problem, they construct models for each of the three problems based on random forests. However, in order to construct such models, a large dataset

containing many runs of the algorithm is needed. Since we do not have hundreds of CIF models at our disposal, this would not work for our purposes. Aside from that, we do not yet know what data to collect to feed into a machine learning model, while the authors have access to many parameters, as their chosen problems are well-studied. Lastly, the problems that the authors have studied are not fixed-point computations, which is what we intend to study. Therefore, this approach is not feasible for this thesis.

Although the works mentioned relate to this work somewhat tangentially, there is no solution yet for our particular problem. Most of the papers mentioned focus on increasing the efficiency of the algorithm instead of detecting when the algorithm should be terminated. The work on process communication graphs does focus on quantifying the effort needed to synthesize a model, but it focuses on intrinsic model complexity instead of accidental complexity due to the settings used. The work that comes closest to what we want is the work on SAT-solving, but unfortunately it does not apply to our work, as the computation differs too much from what we aim to do. Since there is no solution available yet for our problem, we need to do an initial exploration on this topic.

4 Distinguishing good and bad synthesis runs

4.1 Approach

Remember that the user of the synthesis algorithm is presented with a dilemma. When the algorithm does not terminate quickly, should the user wait until it does, or rather terminate the run? Ideally, we would like to communicate to the user how long the algorithm will take. For some algorithms, it is possible to make an estimate of the runtime based on the size of the input. However, the synthesis algorithm is a fixed point computation, and it is not trivial to determine how much time the algorithm will need. Still, we would like to be able to communicate some indication of progress to the user, so that they can decide whether to stop a run of the synthesis algorithm or not.

In practice, we observe that the different settings for the synthesis algorithm can have a large impact on the efficiency of the algorithm. Among others, these settings include the BDD variable order, whether to apply forward reachability to compute reachable states, and the order in which the various fixed point computations are performed. Changing the settings for a given model can make the difference between the controller synthesizing within a second, to becoming infeasible to synthesize [3]. We are interested in the impact that these settings have on the runtime of the synthesis algorithm, independent of the intrinsic model complexity.

We can make that actionable by comparing runs of the synthesis algorithm on the same model with each other. We use different settings for each run. We define a run as *good* when it outperforms most other runs, and *bad* otherwise. Since we are interested in the runtime of the algorithm, we say that a run outperforms another run if it takes less time to terminate. This leads us to the research question: is it possible to determine, having seen a small fraction of a run of the synthesis algorithm, whether the run is good or bad? When the run is over, the user is no longer interested in whether it was good or bad. If instead this classification can be made after having only seen, say, 10% of the run, then the user can use it to terminate the run early and change the settings instead of waiting for a long time.

Since the saturation algorithm lies at the core of synthesis, we need to gather more information on saturation in order to answer the research question. If we gather information on different runs, we may be able to use that information to determine in hindsight whether that run was good or bad. If this turns out to be possible, the next step would be to see if it possible to make this classification based on only a part of the data.

We will gather the necessary information in different ways. In general, when an algorithm is performed, it takes an input, performs some computations on this input, and then gives an output. Also, it is implemented using some data structures. Therefore, we can study algorithms from three perspectives: the input of the algorithm, the internal workings of the algorithm and the data structures used to implement it. Together, these three perspectives provide a complete overview of the algorithm. Specifically applied to the saturation algorithm, this means that we can look at the states BDD (the input) and how it evolves, the transition relations that are applied (the internal workings) and the JavaBDD unique and cache tables (the data structures). We can then look for some indicator of good and bad runs, using these three perspectives.

At this point, we do not know what such an indicator could look like. Therefore, we need to explore the three perspectives as much as possible. However, we do not know in advance what perspective will yield a good indicator. Therefore, we explore the perspectives in parallel using a structured, step-wise approach, to work toward finding an indicator that can predict that a run will be bad, early on in the run. Then, at each step, we can evaluate whether it makes sense to continue exploring this perspective, or whether the other perspectives show more promise. This way, we attempt to find the perspective that gives the most promising indicator, which we can then work out in more detail in Section 5.

In this exploratory study, we use the following six steps.

1. Collect the relevant data, by performing several synthesis runs of the same model, and measuring the appropriate values.
2. Visualize the data, in order to make it insightful.
3. Get an overview of the data, and consider whether this is useful or not.
4. Differentiate between good and bad runs, using the data of a full run.
5. Differentiate between good and bad runs, using only the data of the first part of a run.
6. Conclude whether this data has predictive power, and whether it can predict whether a run will be good or bad.

The first step is to collect relevant data, by performing several synthesis runs. Each perspective considers different data, so this collection step is necessary for every perspective. The next step is to make this data insightful. Since we do not know what indicator we are looking for, visualizations will be helpful toward gaining some sort of insight. When the data has been visualized properly, we can get an overview of the data in step

Name	*Settings*	*Operation count*	*Time (seconds)*
Waterway lock	Good	163,065	0.41
	Bad	60,804,171	4.24
	Worse	914,886,056	72.78
Wafer scanner n1	Good	48,981,942	1.51
	Bad	765,300,632	16.25
	Worse	1,519,536,857	37.18

Table 1: The models and settings used in the exploratory phase, with operation counts. The timings are provided to give an indication of how long it approximately takes to perform a run.

3, and consider whether this data is useful or not. If it is, we can continue with the next steps. If it is not, we can decide to abandon this perspective or to generate different data, depending on the time left. If the data still looks useful, we can continue to step 4, and see if it is possible to differentiate between good and bad runs using this data, after having seen a full run of the algorithm. If it is not possible to classify a run after having seen all of the data, then we cannot hope to do so after having seen only a part of the data. If it is possible to do this classification, we can then see whether it is still possible to do so after having seen a small part of the run, in step 5. If at this point we still pass the test, we can investigate the predictive power of the data further in step 6, since we want to make the data actionable. If a perspective passes step 6, we consider it in Section 5 to construct a predictor using the data gathered using that perspective. All experiments can be found in the artifact that accompanies this thesis [23].

4.2 Benchmark models

Now that the approach is clear, we should define the models that we experiment on. The CIF tooling comes with a set of 18 benchmark models[6] [3]. We use the the benchmark models from Eclipse ESCET v8.0, and experiment on them using Eclipse ESCET v9.0. As the name suggests, these models are used to evaluate performance improvements of the synthesis tool. This set of benchmark models is quite diverse, and contains models that can be synthesized within a second (for example, the `agv` and `adas` models), as well as models that are not yet feasible to synthesize (for example, the `wafer scanner n7` model). For this initial exploration, we need to select a few models that perform really well with one set of settings, and poorly with another set of settings. Since we are limited by the available time, we choose to use only two models, with three different settings. We choose models that both have a large uncontrolled state space, according to [3], and whose settings can easily be altered to create different kinds of runs. Also, we choose one model to run with forward reachability disabled, and another with forward reachability enabled. These models are the `waterway lock` and the `wafer scanner n1` models. Table 1 shows the models and settings used for this exploration, the operation count for each model, and an indication of the time it takes to synthesize the model. The timings shown in Table 1 are measured using a laptop with a 2.3GHz Intel Core i7-12650H processor. Alternatively, the time is measured using the operation count. This operation count is a *platform-independent metric*, that can be used to measure the time, in terms of number of BDD operations [24]. Both models have a good run with relatively few operations, and two increasingly bad runs, as can be seen by the increasing operation counts.

The *good* settings for the models are the ones currently used in the CIF benchmark models. For the `waterway lock` model, this is the default variable order in CIF, with forward reachability turned off. For the `wafer scanner n1` model, the custom variable ordering configuration consists of multiple heuristics applied sequentially. The configuration starts with sorting the variable order, and sets the variable-relation representations to be ordered according to the model's original variable order. Then, it reverses both of these orders. Next, it applies the sliding window algorithm [4], with a window size of 5. Then, it applies the FORCE algorithm [12]. Finally, it applies the sliding window algorithm once more, using the same window size of 5. Note that forward reachability is turned on for this model, and that the fixed point computations order is set to computing reachable states first, then computing the non-blocking states and finally computing the controlled states.

The *bad* and *worse* settings are found using some experimentation, and applying random variable orders. For the `waterway lock` model, the *bad* setting is produced by having the tool generate a random variable order, with seed 57. This seed is found by trying several seeds. The *worse* setting is produced similarly, using the seed 19. For the `wafer scanner n1` model, the *bad* setting is produced by only sorting the variable order, and setting the variable-relation representations to be ordered according to the model's original variable order. Finally, the `worse` setting for this model is again a randomly generated order, generated using seed 2.

The rest of this section consists of three subsections that elaborate on the three perspectives, as well as an evaluation at the end. Section 4.3 elaborates on the input perspective, where the evolution of the states BDD is

[6] Although some of these models come in multiple variants, like the `wafer scanner` and `cat mouse tower` models, so that the total number of models is higher than 18.

tracked. Next, Section 4.4 studies the inner workings perspective. Section 4.5 continues with the data structure perspective. Lastly, the different perspectives are evaluated in Section 4.6.

4.3 Input perspective: the evolution of the states BDD

The first perspective focuses on what happens to the input of the saturation algorithm. Specifically, we aim to track what happens to the BDD representing the states of the system. As shown in Algorithm 3, this states BDD is continuously updated, before the final result is returned. The algorithm constructs intermediate states BDDs, and uses these in the next computation, until a fixed point is reached. By comparing these intermediate BDDs for good and bad runs, we may be able to gain some insight into the difference between good and bad runs.

Step 1: Data collection

As mentioned before, we are looking to keep track of the intermediate BDDs, produced by the saturation algorithm. However, the intermediate BDDs created are only sub-BDDs of the original BDD, of other intermediate BDDs, or of the resulting BDD. For clear terminology, we make the distinction between *layers* and *levels*. A *level* is a term that we have come across in Section 2.3. The root note has level 0, and increases as we travel to the terminals. However, when showing a BDD visually, we often draw the root at the top of the BDD, which is counterintuitive. Therefore, we use the term *layer* for discussing traveling up and down a BDD, where a lower layer is closer to the terminals, and a higher layer is closer to the root. When the algorithm travels down toward lower layers, it alters the states BDD at that particular layer and downward. However, we would like an overview of the entire BDD. The solution here is to impose these lower-layer BDDs onto the original states BDD.

In order to do this, we need the location of the node in the states BDD that the saturation algorithm is currently working on. We can do that by keeping track of a *path*. This path is simply an ordered list containing the `low` and `high` edges that were taken to reach the current node from the root of the BDD. Whenever the saturation algorithm recursively travels down, we append the edge that it took to the list. Whenever the transition relations at that node have been applied, we store this path for later processing. Also, we store the BDD that was newly created by the application of the transition relations, so that we can impose these in the right location.

Now that we have collected the intermediate sub-BDDs of the algorithm, along with the locations they were created in, we can construct new BDDs that represent the entire intermediate states BDDs. To this end, we create a new BDD operation, called *replaceBDD*. This operation takes two BDDs and a saturation path. The first of these BDDs should be the highest up, with its root at the lowest level. The second of these is the replacing BDD. The path determines where in the higher-layer BDD the lower-layer BDD will be placed. The algorithm traverses down the higher-layer BDD, following the path. When the node corresponding to this path is reached, it is instead replaced by the lower-layer BDD. By applying this operation iteratively using all of the stored intermediate sub-BDDs, we can construct the entire intermediate BDDs at a given time.

Step 2: Visualization

Now that we have collected the intermediate BDDs, we need to find a way of visualizing them. The natural choice for this is to make an animation that shows the initial BDD changing over time. This begs the question, how to actually show the frames of this animation? We need to find a way to visually map out the BDDs. Mapping out a BDD is tricky, however. Since we want to show the visual representation of a mathematical concept, there are no hard-and-fast rules on how it should look. There are a lot of choices in where to place a node on the map. Aside from that, the number of possible nodes in a BDD level grows exponentially as you go through the levels, so that if the representation is poorly chosen, higher levels with fewer nodes may not be properly visible.

We would like to represent the BDD using some kind of diagram. A logical choice for this is to use a triangle as the diagram. In fact, we draw a triangle in a rectangular grid of cells, to keep the coordinate system simple. The x-axis of this grid runs from left to right, and the y-axis runs from top to bottom, so that every cell corresponds to an integer coordinate. The top cell of the triangle represents the root of the BDD. The lower cells in the triangle then represent the lower layers of the BDD. We choose the height of the triangle to be equal to the highest level of the BDD divided by two, to filter out the levels that represent new-state variables. This way, we can select the y-coordinate of a node to be equal to half its level. We choose the width of the triangle to be equal to twice the height, as every BDD node has exactly two children.

To fill the triangle, we place nodes in the triangle according to their position in the BDD. This means that we need to assign integer coordinates (x, y) to every node of the BDD, and color the cell at that coordinate. The y-coordinate of a node can simply correspond to its level, divided by 2 to account for the lack of new-state

variables in this diagram. For the x-coordinate, we need to do a more complex computation. Note that we use the term *position* in this computation, which is a number used to compute the final x-coordinate. What follows is the strategy of computing the x-coordinate of node n, followed by the details on every step in the computation.

1. For each parent node of n, compute the position of node n relative to that parent. Positions are computed depending on whether node n is a low or high child of its parent node.
2. Assign to node n the average position of all relative positions.
3. Scale the position of node n so that it fits in the triangle.
4. Round the position of node n to the nearest integer value. Then, shift the position of node n to center the triangle. This results in the x-coordinate of node n.

The first step in the computation is to calculate the position of a node relative to a parent node. We assume that the parent node already has a known position, called p_{parent}. To compute the position of its child nodes from this, we should keep two considerations in mind. Firstly, we should place the nodes in such a way that on a given level, before averaging, no two nodes have the same position. Secondly, levels can be skipped, so that a child node can be placed way lower than its parent. Assuming we do not skip levels, the first issue is overcome by multiplying the position of the parent node by 2. Then, the `low` child can have position $2p_{parent}$, and the `high` child can have position $2p_{parent} + 1$. This way, we can have at most 2^{level} positions per level, which is exactly the amount of possible nodes on that level. When levels are skipped, we essentially have a sub-tree of the BDD that is not represented in the BDD. Since a skipped level in a BDD means that both the `true` and `false` edges lead to the same node, we have two possible positions per skipped level to place the child node. We choose to assign a position to the node "in the middle", where we average over the position obtained by choosing `false` at every level, and the position obtained by choosing `true` at every level, resulting in a position somewhat in the center of the skipped sub-tree. Using this computation, we can assign a position to every node, based on the position of a given parent node.

The second step in the computation of the x-coordinate of node n is to calculate the average position relative to all parents. Recall that node n can have multiple parents due to node sharing in BDDs. To compute the average position of node n, we must make sure that we know the position of all parent nodes of node n. Therefore, we must traverse the BDD breadth-first, starting from the root. We assign position $p_{root} = 0$ to the root. When encountering a new node n, we first compute its position by averaging over the relative positions compared to its parent nodes. Then, we use this position to compute the relative positions of its child nodes, as described in the previous step. Then we continue traversing the BDD, until the terminal nodes are reached. This way, we can assign positions to all nodes in the BDD.

The third step in the computation is to scale the positions of all nodes so that they fit within the triangle. This is necessary as there are more positions in a given BDD layer than there are cells in the triangle at that layer. As seen from the root, which has level 0, we have at most 2^l positions in a given level. However, we have a width of $2l$ cells at this layer (excluding level 0, which has a width of 1 cell in order to be able to place the root). This means that in order to fit all positions in the triangle, we need to scale every position by $\frac{2l}{2^l}$. As a consequence, the positions are no longer integers, which will be addressed in the next step. The result of this scaling is that every position is mapped to fall within the bounds of the triangle.

The final step in the computation is to convert the position of node n to a concrete x-coordinate in the grid. Firstly, since the grid is discrete, we round the position of the node to the nearest integer. If done for all nodes, this in principle results in a triangle. However, this triangle is aligned to the left of the grid. Therefore, we shift every position to the center, so that the tree is centered in the grid. Now, the position of a node corresponds to a concrete x-coordinate in the grid, which completes the computation.

As mentioned earlier in the third step of the computation, the positions of the nodes have to be scaled to fit the triangle. This means that the positions of multiple nodes can be mapped to the same cell. In order to try and visualize how many nodes are mapped to each cell, we could color the cell differently based on the number of nodes in that cell. This way, we can represent densities of nodes in the visualization. Since the lower layers of the triangle can contain exponentially more nodes per cell, we could scale the color down exponentially too. However, the experiments reveal that very few nodes are mapped to each cell. Therefore, we decide to give each cell the same color. Figure 2 shows a visualization for the initial BDD of the backward reachability analysis of the `wafer scanner n1` benchmark model, with *good* settings. Note that there are many black cells in this image, as the predicate does not require many BDD nodes to represent it.

Now that individual frames of the animation can be constructed, we can construct the entire animation by showing these frames one after the other. In order to give insight into where the saturation algorithm is applying transitions, we color the sub-BDD that has changed this frame differently (white pixels) from the rest of the BDD (orange pixels). Figure 3 shows a few frames of the animation for the first forward saturation pass

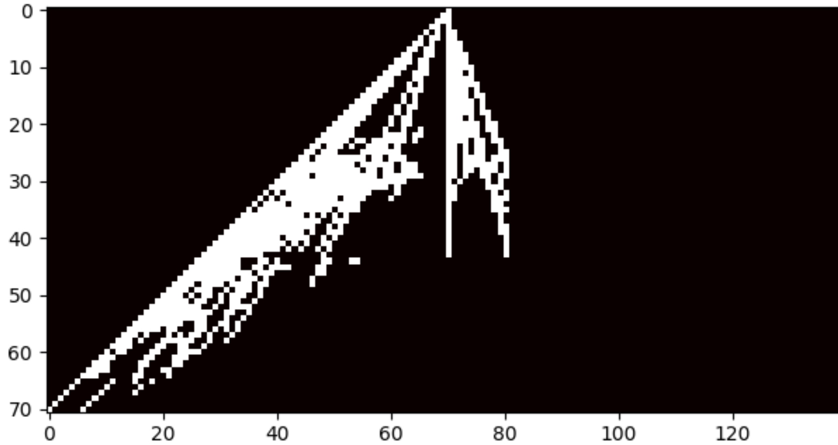


Figure 2: The visualization for the initial BDD of the first backward reachability computation of the `wafer scanner n1` benchmark model, with *good* settings. Note that all cells have been made white, instead of performing the density calculation. The y-axis represents the level of a node, while the x-axis represents the location of the nodes relative to their parents.

on the `wafer scanner n1` benchmark model, with *good* settings. It can be seen that the saturation algorithm creates more nodes in this case.

Step 3: Overview of data

When analyzing the animations, it seems as though some of the runs show a "blow-up". In these runs, a slim initial BDD explodes into a wide BDD, as can be seen in Figure 4. This figure shows several frames of the first backward saturation computation on the `waterway lock` benchmark model, with *bad* settings. The good settings do not seem to show this same phenomenon, however, as can be seen in Figure 5. If this is indeed an indicator of a bad run, then this blow-up could be the moment to signal the user that the run has gone bad. Therefore, it is worth investigating this phenomenon further.

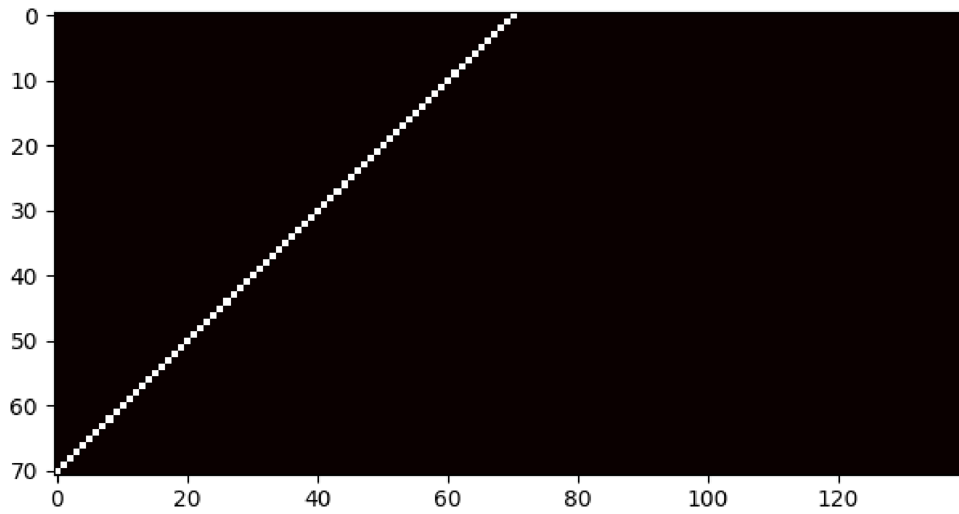
Step 4: Differentiating good and bad runs

Now that this blow-up has been identified as a possible indicator of a bad run, we need to investigate further whether it actually is an indicator of a bad run. As an animation can be difficult to follow, we convert it to a static image. Since we want to focus on the blow-up, we should investigate the number of occupied cells over time. To this end, we plot the percentage of occupied cells in the grid over time, as in Figure 6. However, from these plots, it becomes clear that this correlation may not be that strong. In fact, the *worse* run show less of a blow-up than the *bad* run. Even more, the *worse* run shows a steady decrease of occupied cells in the first part of the plot. It could still be the case that the blow-up indicates a bad run, but at this point it is not convincing enough to conclude this. Since the other benchmark also does not show a strong correlation, we decide to stop exploring this perspective, and focus on the other perspectives instead.

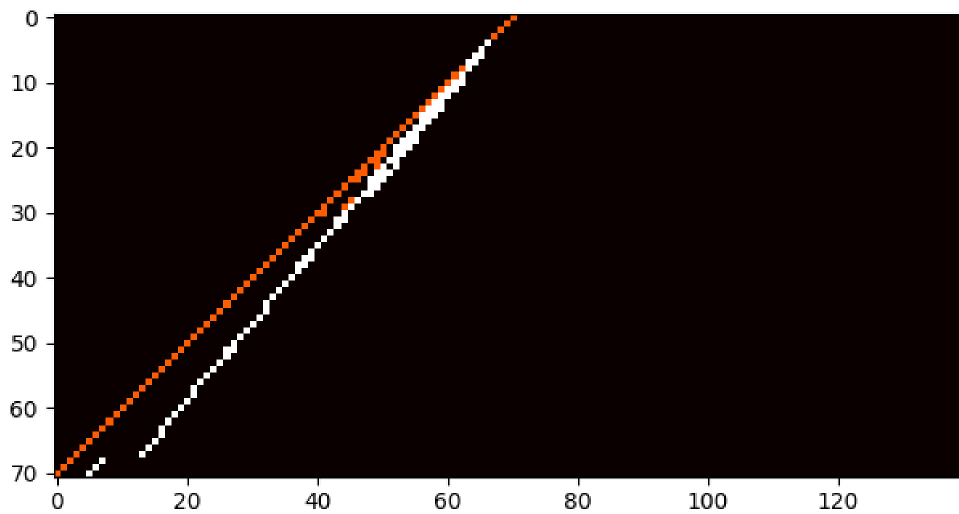
4.4 Inner workings perspective: the application of transition relations

The second perspective focuses on the inner workings of the saturation algorithm. As seen in Algorithm 3, the core of this algorithm is a loop where groups of transition relations are applied until the given node of the states BDD is saturated. As mentioned in Section 2.5, the algorithm moves both up and down through the layers of the states BDD, as well as back and forth through the list of transition relations. These two are not independent, however, as transitions are applied at the level of their lowest variable, which are determined before saturation starts.

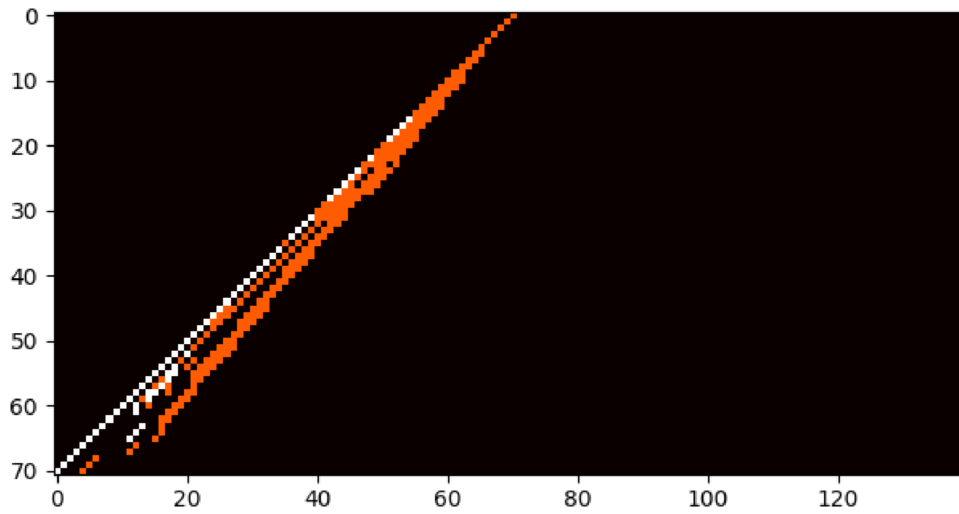
The relation between transitions and variables can be visualized using the *saturation matrix* that is already present in the synthesis tool. When the user chooses to enable debug output, they will see this saturation



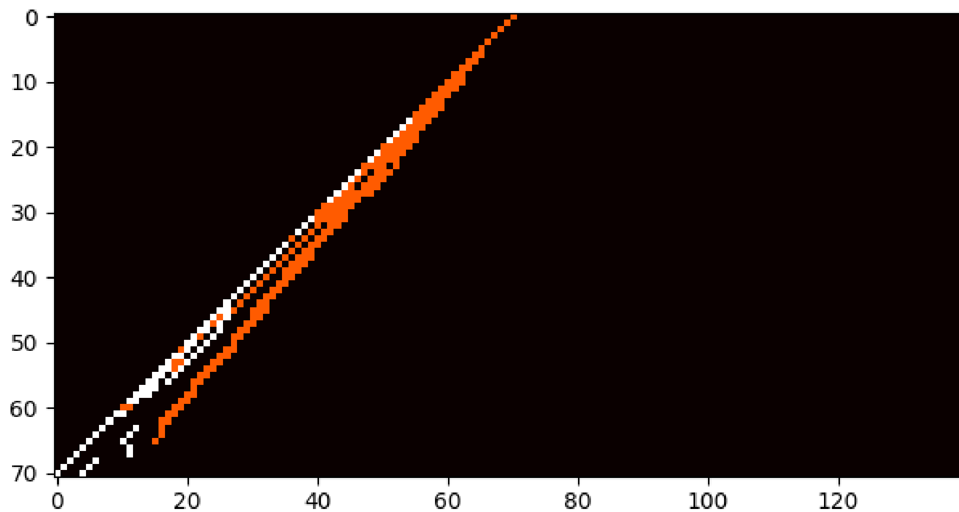
(a) The initial BDD. Note that it is almost a straight line. Also, this BDD is different from the one shown in Figure 2, as this is the BDD prior to the forward reachability computation instead of the backward reachability computation.



(b) The BDD after a few frames of the animation. Several other nodes have been created by the algorithm. Note that the algorithm is working on the right side of the BDD (indicated by the white cells).

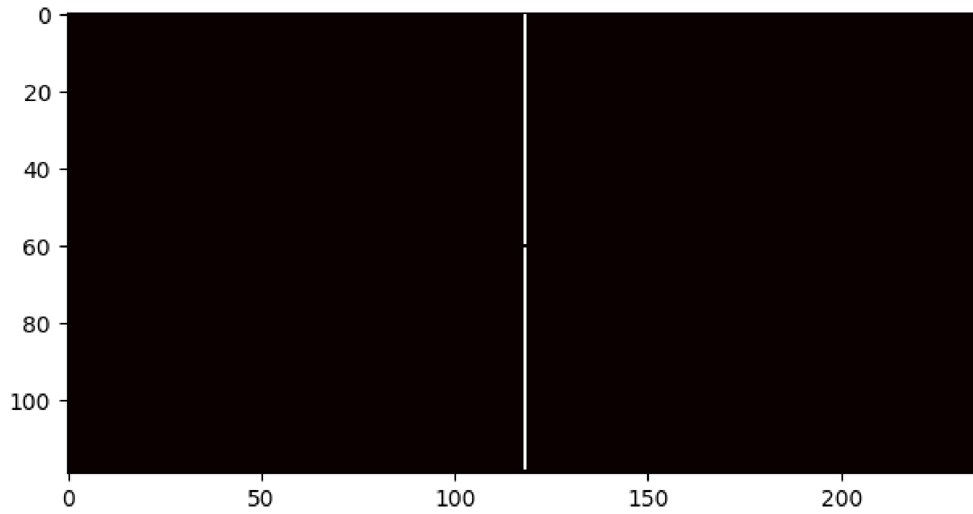


(c) After even more operations, it can be seen that even more nodes have been created. The algorithm is now working on the left side of the BDD.

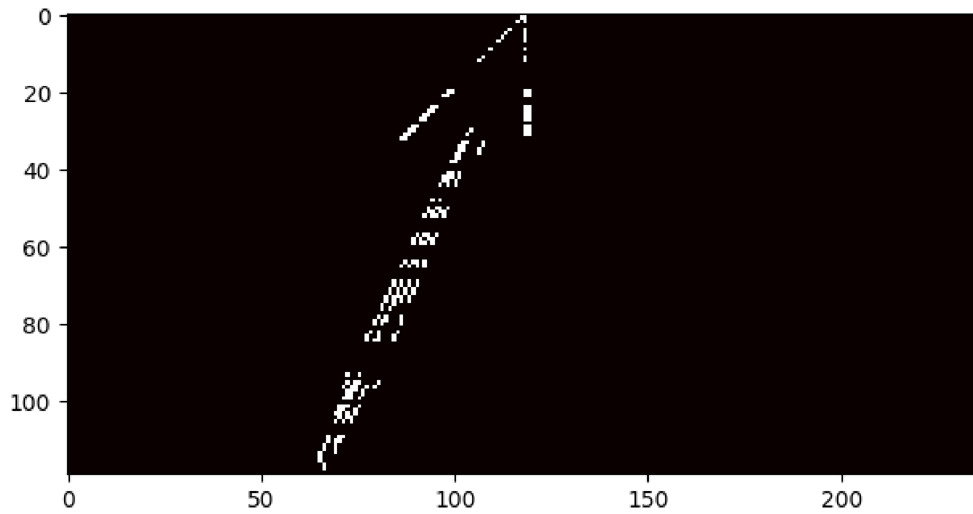


(d) The final BDD. Note that the BDD has expanded, but far from the entire triangle is occupied. The white cells indicate the final sub-BDD that the algorithm has worked on.

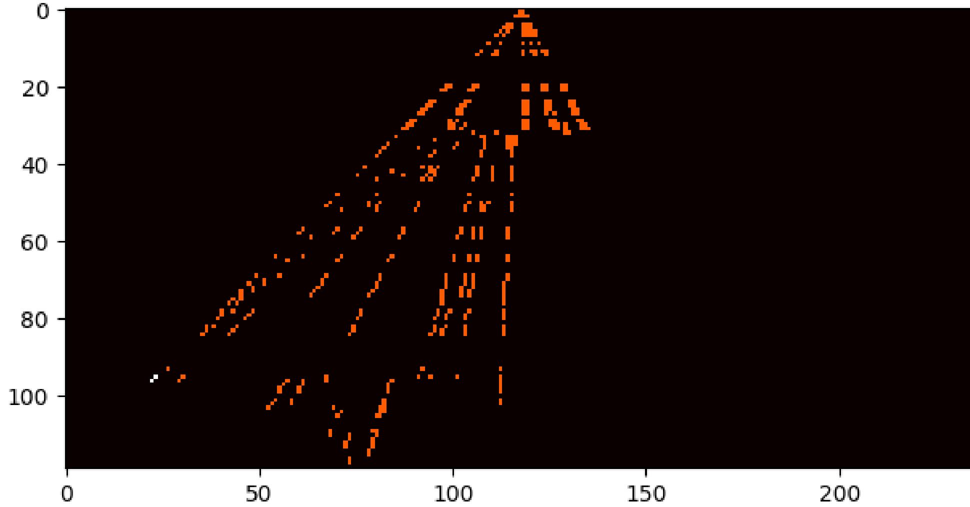
Figure 3: A few frames of the animation for the first forward reachability calculation for the `wafer scanner n1` benchmark model, with *good* settings. The visualizations consist of orange and white cells. Here, a white cell means that this is a cell that saturation is currently working on, while the rest of the cells are orange.



(a) Visualization of the initial BDD. This is almost a straight line from top to bottom.



(b) Visualization of the BDD after several operations have been performed. There are clearly more nodes in this frame than in the initial frame.



(c) A few frames later, even more nodes were created rapidly. Note the algorithm is working on a small part of the BDD on the bottom-left.

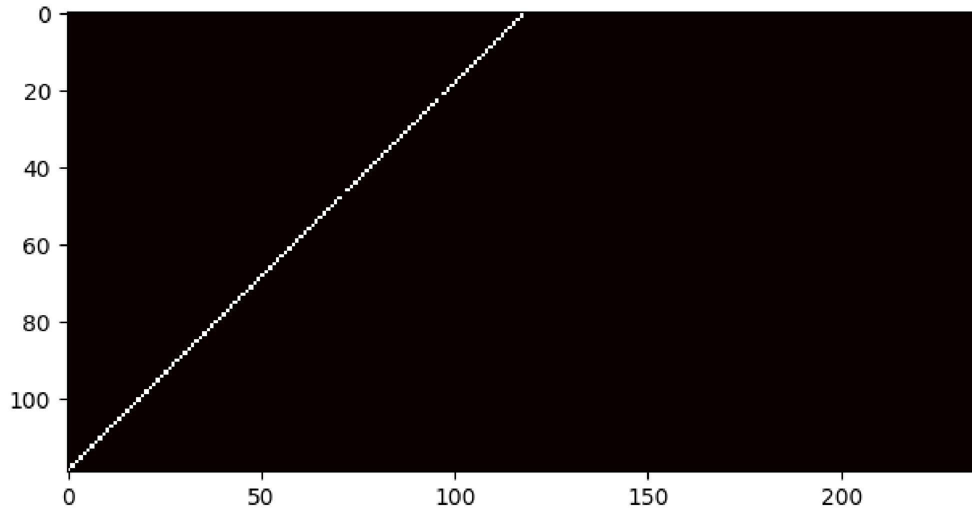
Figure 4: A backward reachability computation for the `waterway lock` benchmark model, with the *bad* settings. There appears to be a blow-up of nodes, where many nodes are created rapidly.

matrix before each reachability computation. The y-axis of the matrix represents the transition relations, and the variables are represented by the x-axis. Note that the saturation matrix prints information at the BDD level, so that these variables correspond to BDD variables. Therefore, in the matrix, old-state variables and new-state variables are interleaved. The cells of the matrix correspond to r or w values, which stand for *read* (old-state) and *write* (new-state) variables. If a cell contains neither r nor w, then the x-coordinate of the cell corresponds to a variable that is not contained in the transition relation represented by the y-coordinate of that cell.

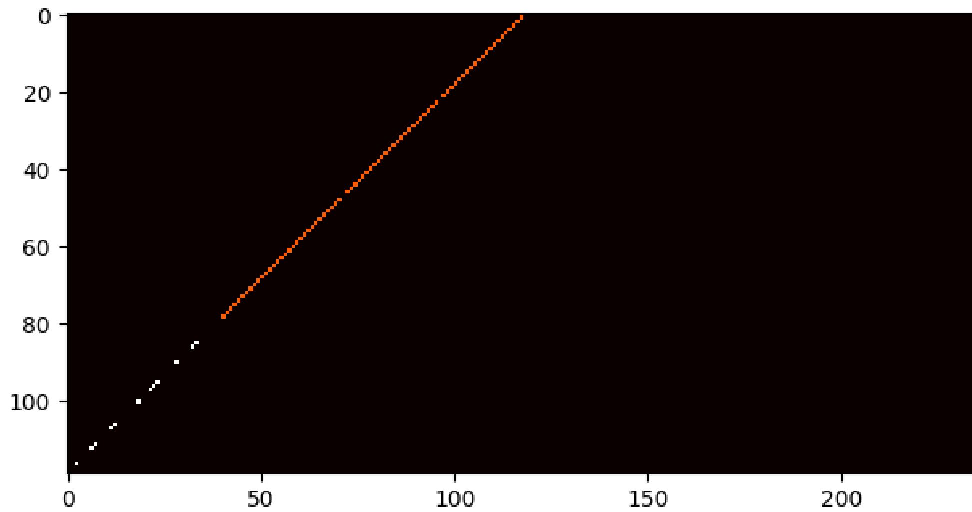
Using the saturation matrix, the user can already get an indication on how efficient the saturation algorithm will be. If the matrix is a diagonal, then no variable appears in more than one transition relation. This means that when the saturation algorithm saturates a node at a given level, it does not alter the nodes at the lower layers. As a result, when moving back toward lower layers, the result of the relational products are likely still in the operation cache. This makes the algorithm work efficiently for these kinds of saturation matrices. In general, off-diagonal entries impact performance. Most notably, if the matrix contains vertical lines, where the same variable is read and written to across multiple transition relations, the algorithm will usually perform poorly. This is due to the fact that writing to a variable invalidates earlier reads of that variable, so that the nodes corresponding to that variable have to be saturated again. Horizontal lines on their own are less of an issue, as these indicate that a certain transition relation contains many variables. The odds that other transition relations also act on a subset of these variables increase, however, such that performance can still suffer from these horizontal lines.

Transition index	*Event name*	*Variable 1*	*Variable 1'*	*Variable 2*
1	button3.u_push	r	w	
2	button3.u_release	r	w	
3	culvert_N.c_enable	r		r

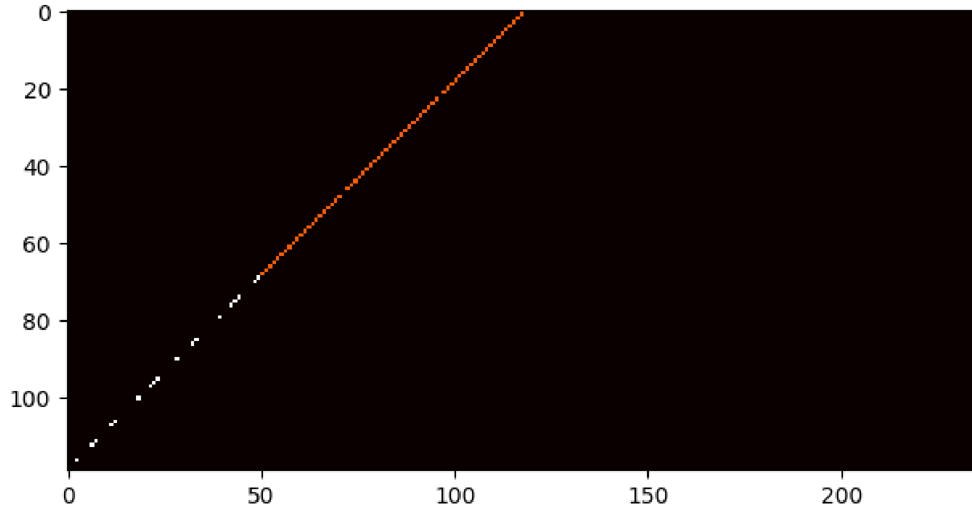
Table 2: A small part of the saturation matrix of the `waterway lock` model, using the *good* settings. The matrix shows events, or transitions (identified by their CIF events), on the y-axis, and variables on the x-axis. An r indicates that the variable is read in the corresponding event, a w indicates that the variable is written to in that transition, and an empty cell means that that variable is not part of the transition.



(a) Visualization of the initial BDD for a run with *good* settings. Note that the BDD is very slim, as it is almost a straight line.



(b) Visualization of the BDD after several operations have been performed. The BDD is still very slim, and few additional nodes seem to have been created.



(c) At the end of the saturation algorithm, the BDD is still very slim. The blow-up phenomenon cannot be observed here.

Figure 5: A backward reachability computation for the `waterway lock` benchmark model, with the *good* settings. The BDD stays close to its original size, and a blow-up cannot be observed.

Intuitively, the algorithm starts at the bottom-right of the matrix, and ends at the top-left, as it first applies the transition relations near the lower layers of the BDD, and moves up when nodes are saturated. This means that perhaps this application of transition relations reveals something about how fast the algorithm works. By studying the application of these transition relations, we aim to find some indication of the progress that it is making.

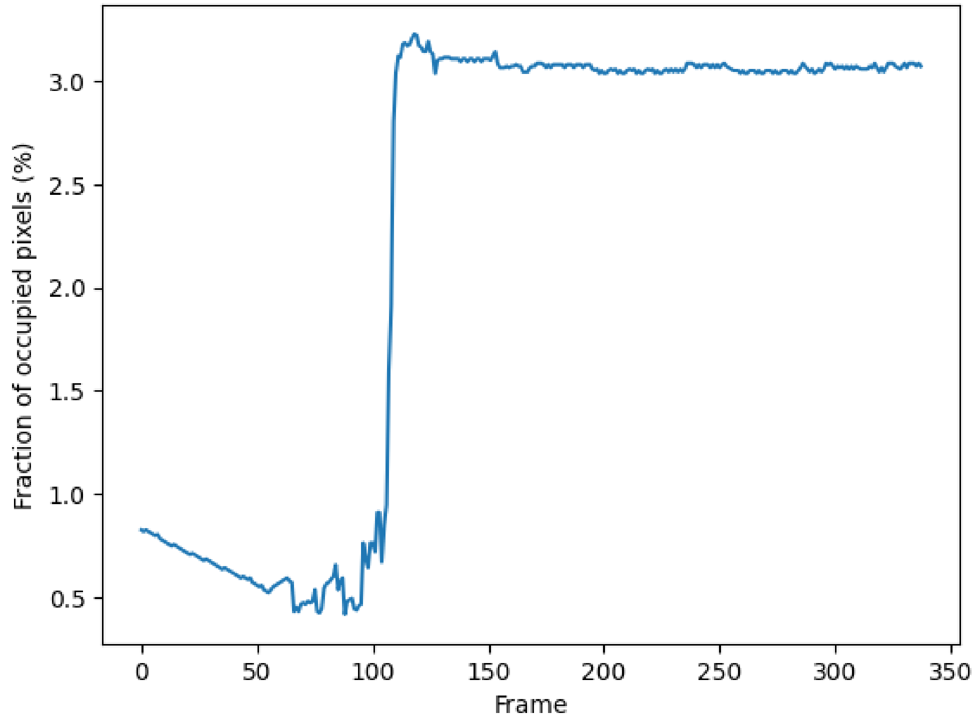
Step 1: Data collection

The data used in this perspective is the application of transition relations over time. This can simply be collected by recording which transitions are applied at which time, measured in the operation count metric. Specifically, JavaBDD has a `saturationCallback`, which is called after every application of a transition relation during the execution of the saturation algorithm. We use this callback to record the specific transition relation that was applied and the operation count at which is was applied. We also store the support of this transition, which is the set of all variables that occur in this transition relation. We write these results to a file for further processing.

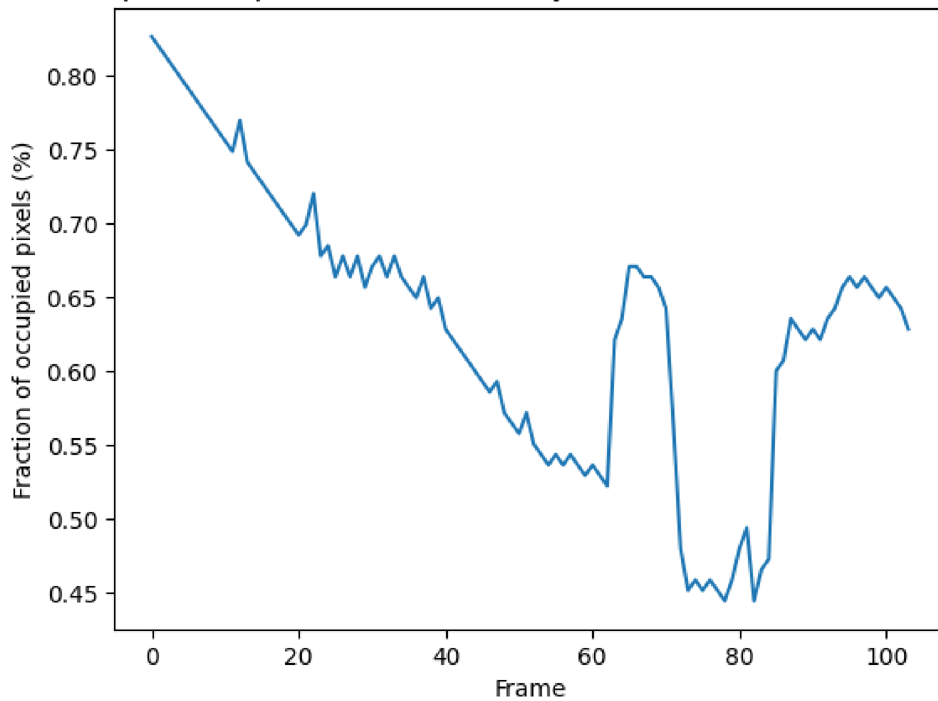
Step 2: Visualization

To visualize the data, we take inspiration from the saturation matrix. The collected transition data can be used to animate the saturation matrix, to get a feeling for how the saturation algorithm actually "moves to the top-left". The canvas for this animation is a heatmap, consisting of cells that can be colored. This heatmap is structured in the same way that the saturation matrix is, where the columns represent the variables and the rows represent the transitions. The application of a transition can then be represented by coloring the cells at the coordinates corresponding to the correct transition and associated variables. The other cells remain black, so that the matrix itself is as clear as possible. When displaying the heatmaps of the applied transitions in order of application, we get an animation that represents the saturation algorithm.

This animation does not provide much insight yet, however. When showing a single transition at a time, the animation moves way too quickly, and there is no smooth transition between frames. Therefore, we adapt the animations to use a window of 100,000 operations per frame, instead of a single application of a transition relation. This window size was deduced experimentally so that each frame shows multiple applications of



(a) The evolution of the occupied pixels over time, in the animation of the first backward reachability computation of the `waterway lock` model with *bad* settings.



(b) The evolution of the occupied pixels over time, in the animation of the first backward reachability computation of the `waterway lock` model with *worse* settings.

Figure 6: Comparison of the blow ups of occupied pixels for the waterway lock model, for the *bad* and *worse* settings.

transition relations. If a transition is applied multiple times in a frame, then the color of the corresponding cells is made brighter. This produces a fade-out effect, which gives a clearer image of the algorithm. Figure 7 shows several frames of the animation made for the `wafer scanner n1` model, with *good* settings. Specifically, the frames are taken from the initial forward reachability calculation.

Step 3: Overview of data

The animation indeed shows something interesting. When looking at the animation for a run with a nearly diagonal saturation matrix, such as for the `waterway lock` model with *good* settings, the animation moves to the top-left quickly. Here, the saturation algorithm does not have to do much work when it moves back to the bottom-right, since the transition relations mostly act on different variables. For models with worse settings, the animation takes way longer, and it can be seen that it oscillates a lot. This leads to the idea of some sort of *velocity* for the saturation algorithm. A higher velocity would mean that the algorithm moves to the top-left quicker, and a lower velocity would mean that it has slowed down. This slow down could then indicate that the current run is indeed a bad run, at which point the user can be informed to try different settings.

Step 4: Differentiating good and bad runs

To investigate this idea further, we turn the animations into single plots, that show the entire run. These plots, seen in Figure 8, show the evolution of the transitions that are applied and their corresponding top-layer variables for runs of the `waterway lock` model. Then, to determine the velocity, we fit a linear, least-squares fit to this data, and define the velocity to be the slope of this line. Now, the *good* runs indeed appear to show that the velocity is negative, which corresponds to the animation moving to the top-left. The *bad* runs appear to have a nearly flat fit, which could be the indicator that we are looking for. Since the results still appear promising, we continue to the next step, and try to differentiate between good and bad runs early.

Step 5: Differentiating good and bad runs early

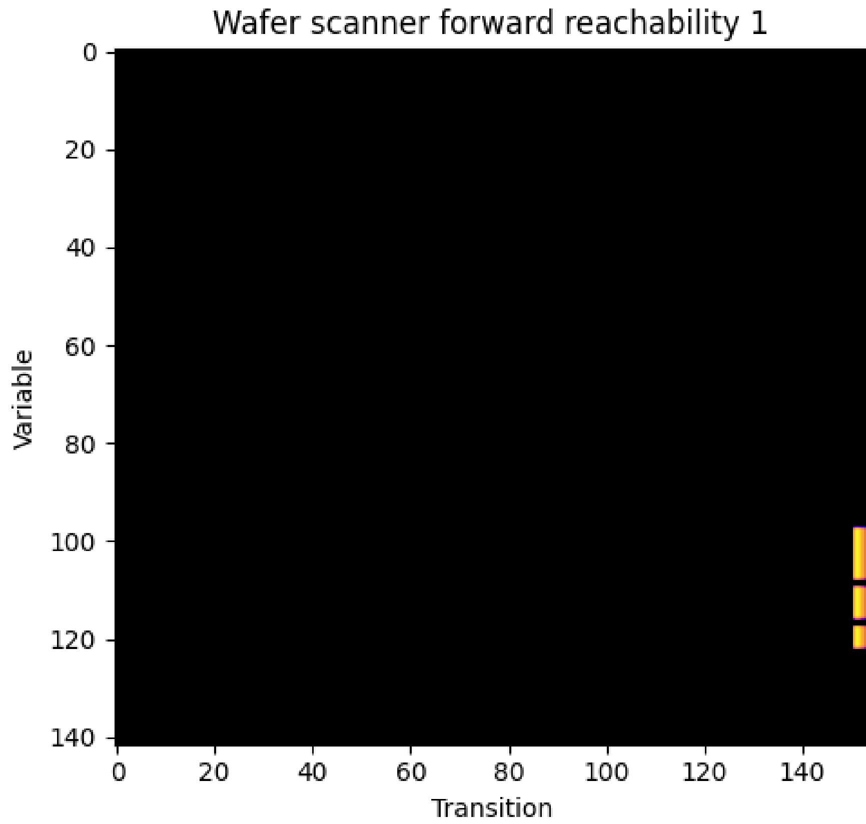
To be able to differentiate between good and bad runs early, we can look at the evolution of the velocity over an entire application of the saturation algorithm. The idea here is to find some sort of tipping point for the velocity, a point where the velocity moves from negative to positive, to indicate a point where the run is no longer good, or even remains bad for a longer period of time. If good runs result in negative velocity, then perhaps bad runs start with negative velocity, and flip to positive velocity at some point in the run. This tipping point would then be the time to communicate to the user that they should probably not wait for the algorithm to terminate. To do this, we divide the data into 100 segments, where segment 1 contains the data of the first percentage of the run, segment 2 contains the data of the first two percent of the run, etc. Then, for each segment, we compute the velocity as above, and plot this. The result is shown in Figure 9.

Unfortunately, the plots do not show much reason to continue this direction. The velocity does not seem to have a tipping point, and is instead all over the place. Bad runs can have more positive or more negative velocities than good runs, with little difference between a good and a bad run. The velocities for the `wafer scanner` runs also show little promise. Therefore, we make the decision to abandon this perspective, and focus on the other perspectives instead.

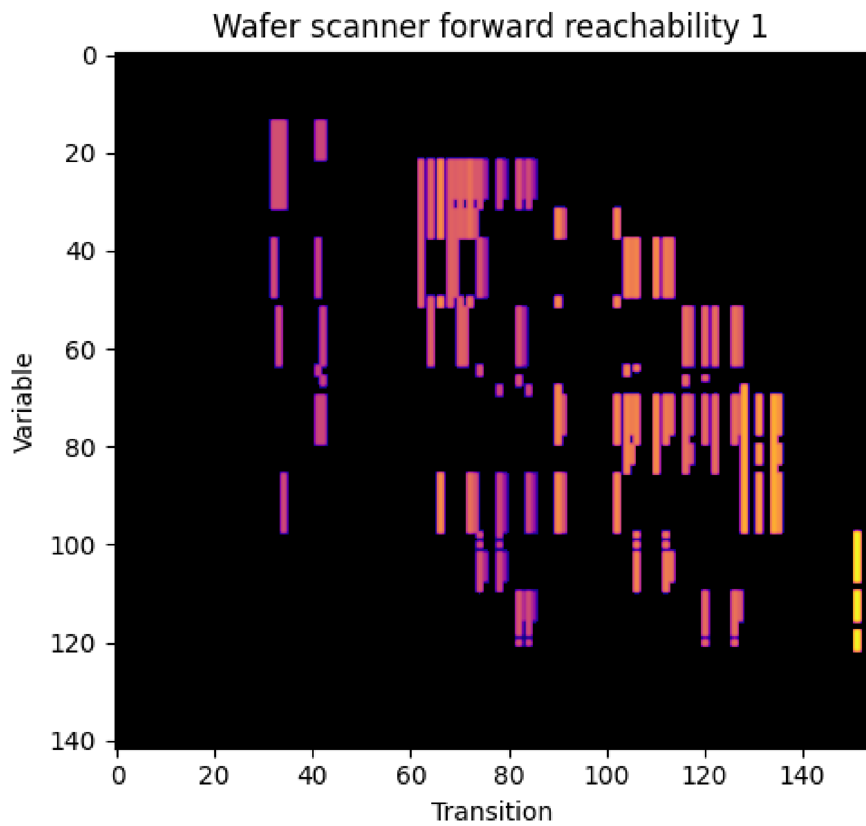
4.5 Data structure perspective: implementation in JavaBDD

The third perspective focuses on the implementation of the saturation algorithm in the JavaBDD library. As mentioned in Section 2.6, the library uses a unique table to store nodes, and an operation cache table to store results of previously performed operations. As mentioned earlier, this cache table is lossy. This means that some computations may have to be performed multiple times, as an operation result that was lost has to be computed again.

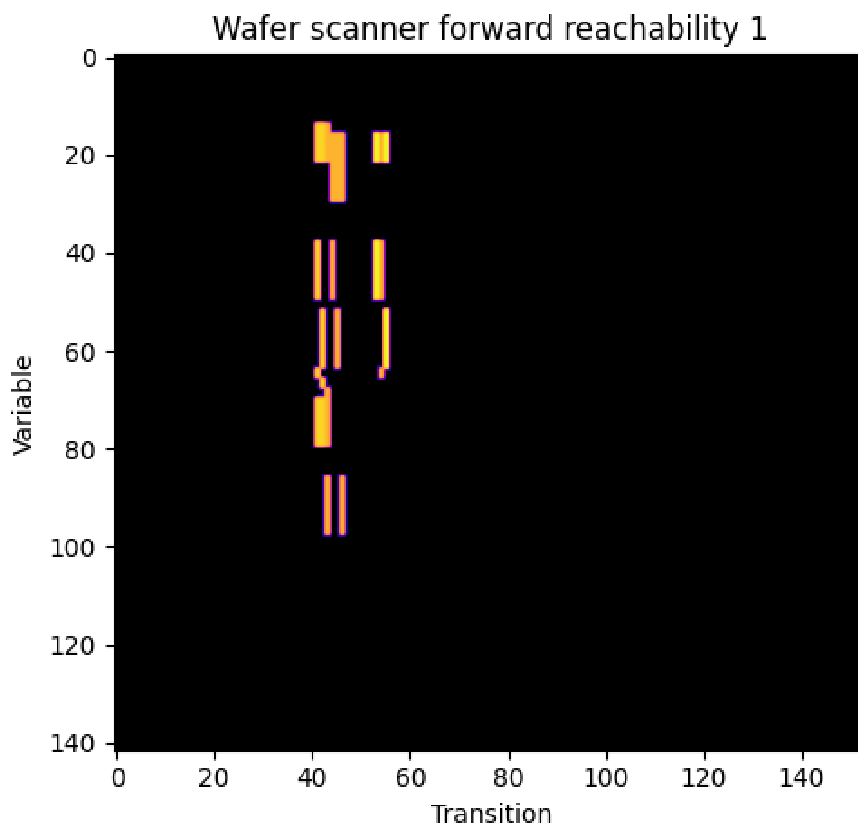
The hypothesis here is that the saturation algorithm specifically suffers from this lossiness. As mentioned before, the saturation algorithm repeatedly applies variants of the `relnext` or `relprev` operations until a fixed point is reached. Consider the following scenario: one of the `relnext` operations produces a certain BDD node *node*. One of the next `relnext` operations, which uses a (different) transition relation that also contains nodes at that level, causes *node* to no longer be a part of the representation, so that it is no longer referenced. It is not removed from the node table until the next garbage collection. Since *node* is not present in the final states representation, we call it an *intermediate* node. Now assume that many of these intermediate nodes are created, and at this point, garbage collection starts. All of the no-longer-referenced nodes are removed from the unique table, and their operation results are removed from the cache table. However, as the saturation algorithm can move back to that same level, it may happen that the exact same nodes that were just deleted, are re-created. At this point, however, all of the associated cache entries were lost, so that all computations have to be repeated.



(a) One of the earlier frames in the saturation animation of the wafer scanner n1 model. Here, a transition on the bottom-right is applied.

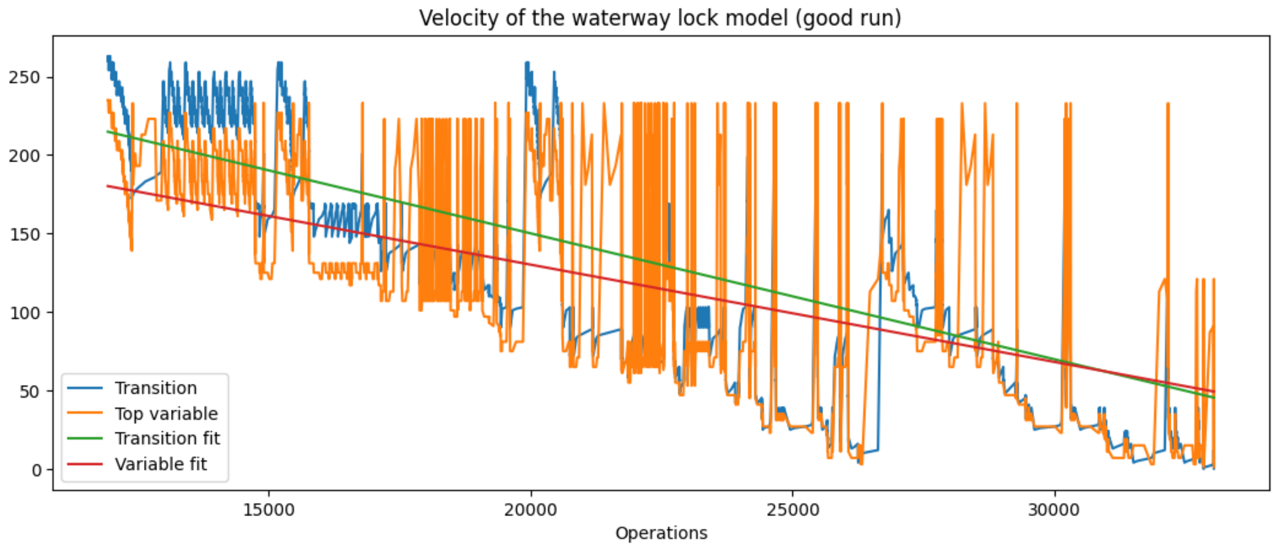


(b) A few frames later, more transitions are visible. The earlier transitions slowly fade-out.

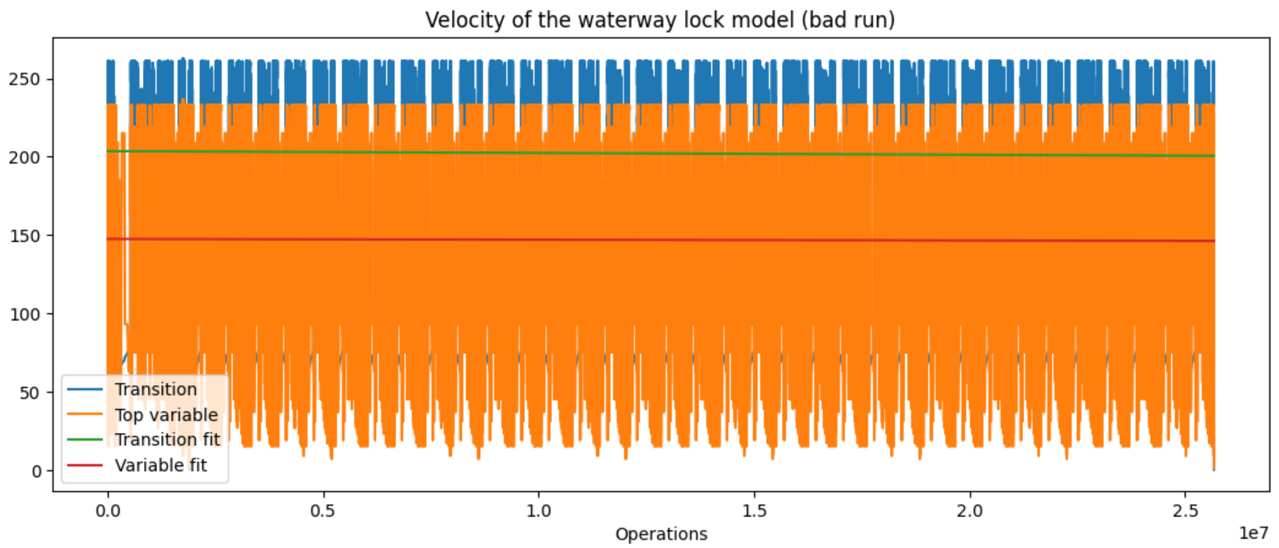


(c) Even later, only transitions on the top-left are applied in the current window. The other transitions have faded out.

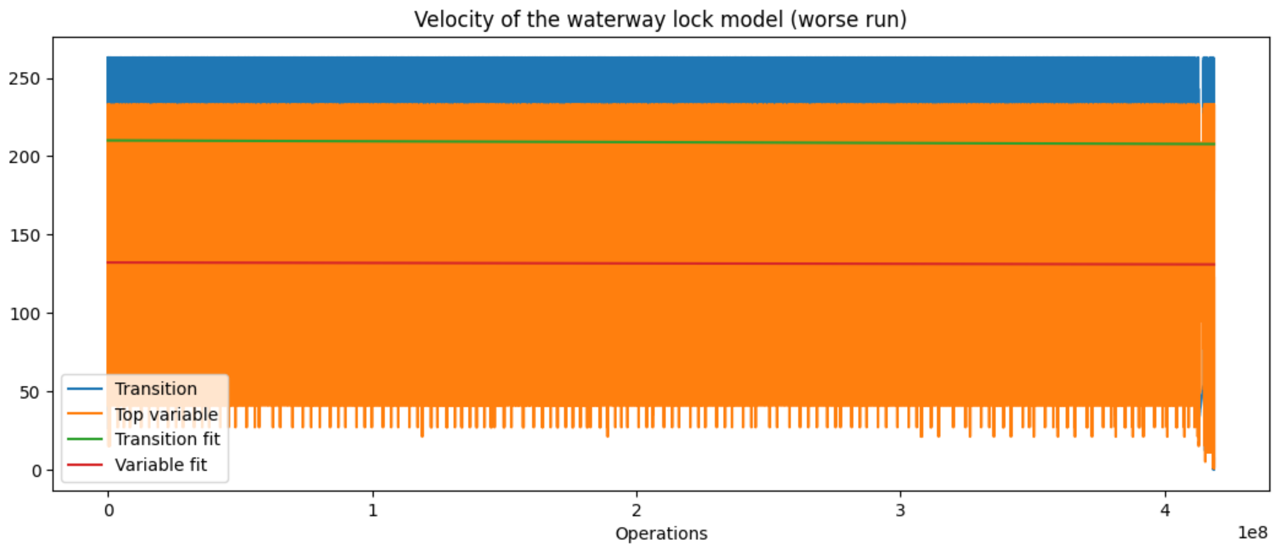
Figure 7: Selected frames from the animation of the initial forward saturation of the `wafer scanner n1` model, with *good* settings. Here, it can be seen that the saturation algorithm first applies the transitions on the bottom-right, and over time moves to the top-left.



(a) The velocity plot for the `waterway lock` model, with *good* settings. Note that the velocity is clearly negative.

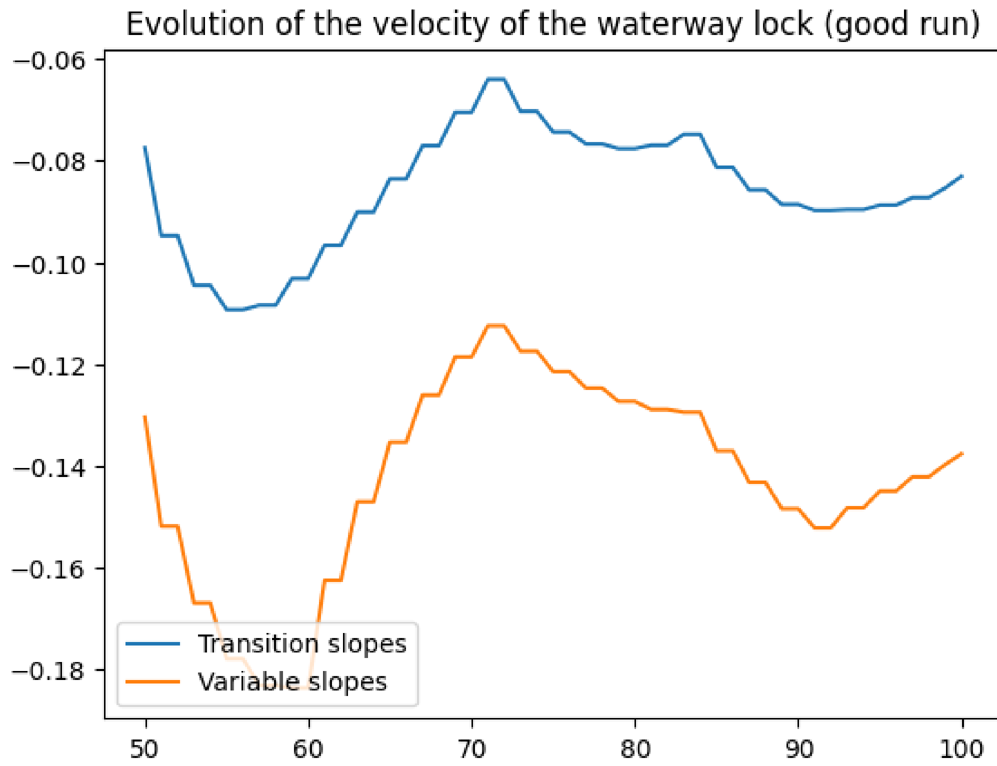


(b) The velocity plot for the `waterway lock` model, with *bad* settings.

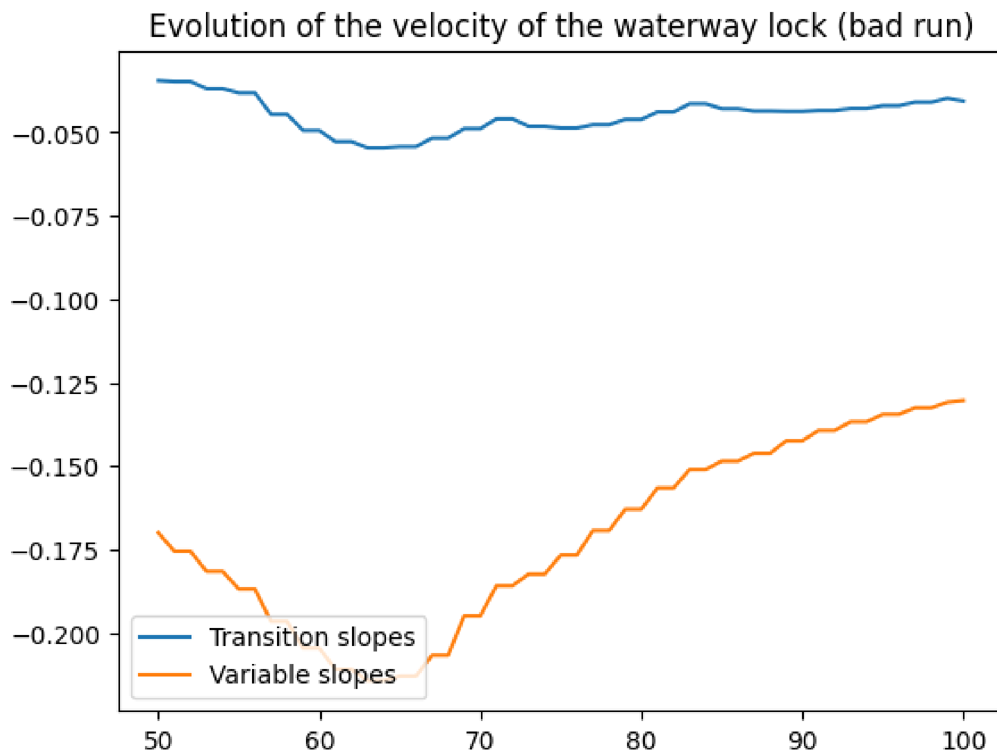


(c) The velocity plot for the `waterway lock` model, with *worse* settings.

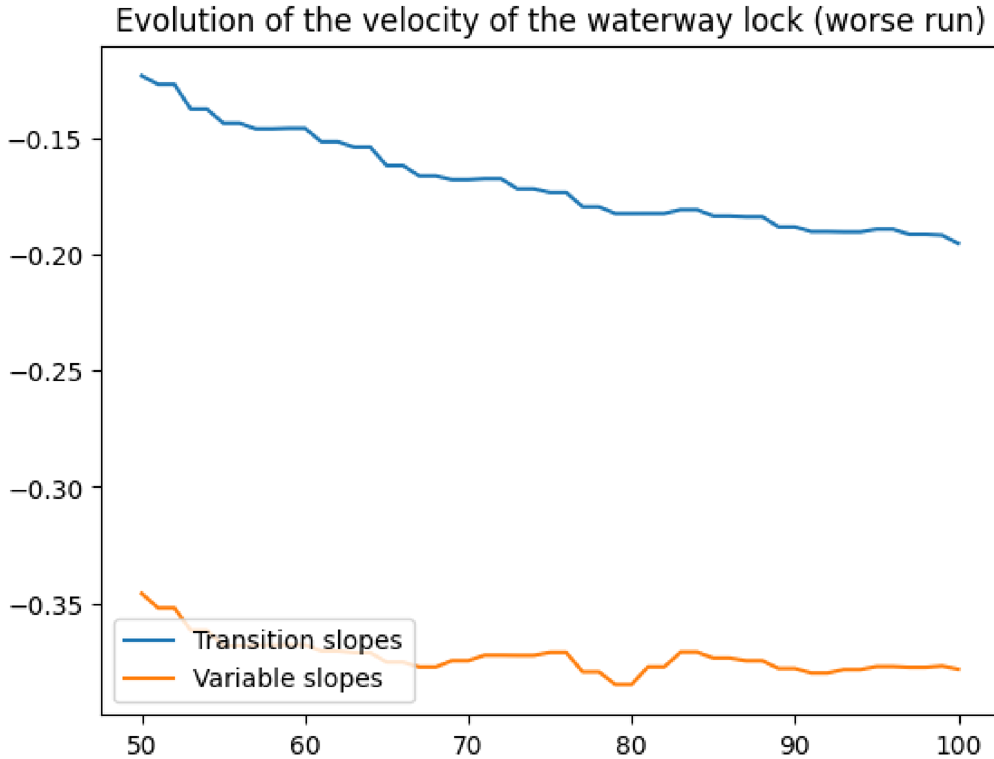
Figure 8: Velocity plots for several runs of the `waterway lock` benchmark model.



(a) The evolution of the velocity of the `waterway lock` model, with *good* settings.



(b) The evolution of the velocity of the `waterway lock` model, with *bad* settings.



(c) The evolution of the velocity of the `waterway lock` model, with *worse* settings.

Figure 9: The evolution of the velocities of several runs of synthesis, for the `waterway lock` model.

When many duplicate computations are performed during a run of the saturation algorithm, a lot of time is wasted, as the results of these computations were already known at some point. Even worse, when a computation has to be performed again, it could produce new cache entries, which overwrite other important cache entries, which leads to more duplicate computations in the future, so that the problem intensifies over time. Therefore, the creation of duplicate nodes or the occurrence of duplicate computations may be indicators of a bad run.

Step 1: Data collection

In order to investigate this problem, we need to collect data on duplicate computations and duplicate node creations. To do so, we need to keep track of all nodes and cache entries that have been created during the synthesis algorithm, and upon creating new ones, check whether it is a duplicate. There is a problem with this approach, however. The JavaBDD uses the location of a node in the unique table as the identifier of that node. However, when the node is removed from the table, another node could take its place. If this happens and the node is re-created, it will then be placed in a different position in the table, and therefore have a different identifier. This means that this identifier cannot be used to identify nodes between garbage collections.

To solve this, we use a unique identifier for a node based on the structure of its child nodes, independent of the node table. One way to do this is to use a pairing function $P : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, that uniquely encodes a node using its two child nodes. However, it quickly becomes clear that such an approach becomes infeasible: there are so many possible BDD nodes that in order for the identifiers to be unique, the identifiers themselves require megabytes of memory to store. A different solution is to accept that different nodes can have the same identifier. Now, an identifier can be constructed by hashing. For the two terminals, `false` and `true`, we take the hash values of 0 and 1 respectively. For an internal node, the identifier can be calculated as $id = hash(l, id_{low}, id_{high})$, where l is the level of the node, id_{low} and id_{high} are identifiers of the low and high children, and $hash$ is some hash function. Note that only hashing the identifiers of the children is not enough, as nodes on different levels can have the same children. Since this calculation only depends on the structure of a node, it will be the same regardless of its location in the unique table. We extend the encoding of the entries of the unique table, so that each node consists of 7 integers instead of 5. The remaining 2 integers are used to store the 64-bit identifiers. It should be noted that hash collisions can occur. However, given the fact that 64-bit hashes are used, and the number of node creations is typically way smaller than 2^{64}, it seems unlikely that many hash collisions occur. Figuring out how many collisions occur exactly remains future work.

Now that every node has an identifier, it is possible to measure how many duplicate nodes are created, by

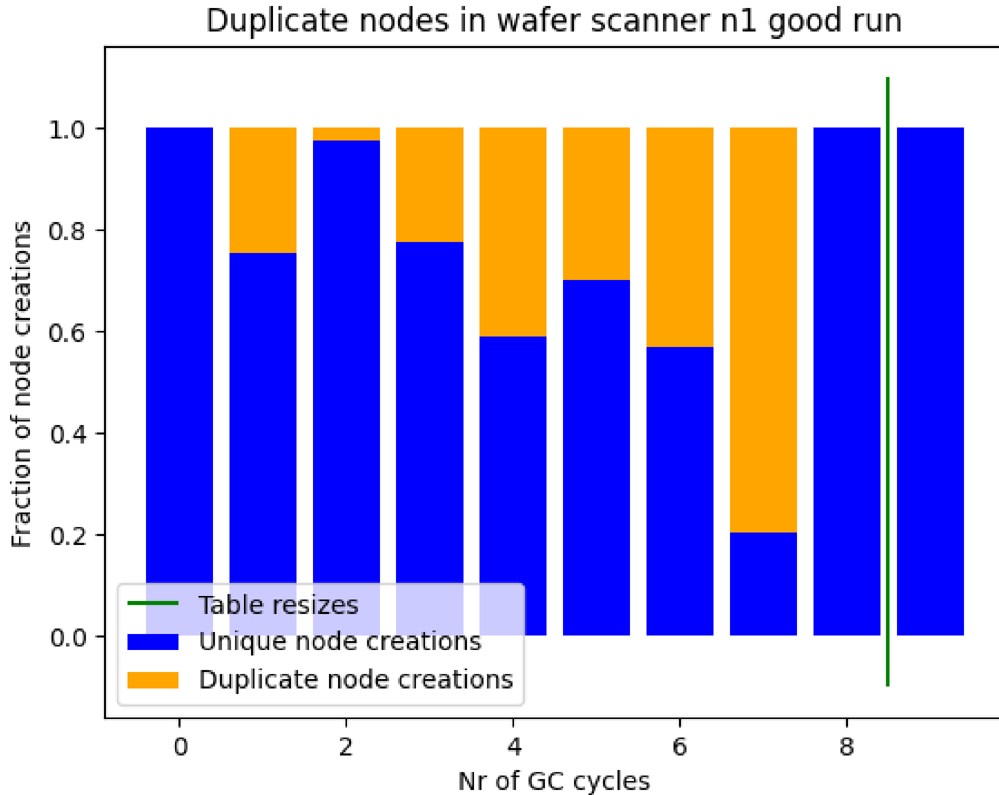


Figure 10: Duplicate node creations during a run of the synthesis algorithm for the `wafer scanner n1` model, with *good* settings.

keeping track of all created identifiers. When a node is inserted into the node table, its identifier is calculated. Then, we check whether that identifier is already in the set of previously created identifiers. If not, we add the identifier to the set. If it was already in, we increase the counter that keeps track of duplicate node creations. Also, we keep track of the total number of node creations, so that we can compute the ratio of unique and duplicate node creations. To keep this data insightful, we store these numbers per garbage collection cycle, as the garbage collections are a major cause of the problem.

Additionally, we can use the identifiers to keep track of duplicate cache entries. A cache entry consists of an operation identifier, and identifiers for the input nodes, as well as the output node. To measure the amount of duplicate computations, we employ the same strategy as for measuring duplicate nodes, by keeping track of a set of previously performed computations and their results. Here, we use a separate counter for each operation. This way we can also get an overview of which operation results are being recomputed.

Step 2: Visualization

Figures 10 and 11 show the duplicate node creations during a run of the synthesis algorithm for the `wafer scanner n1` benchmark model with the *good* and *worse* settings. Here, the blue bars represent the fraction of unique node creations, and the orange bars represent the fraction of duplicate node creations, per garbage collection cycle. The table resizes are represented with green lines, in between garbage collection cycles. It can be seen that the *worse* settings produces a large fraction of duplicate nodes, while the *good* settings produces mostly unique nodes. Note that the table resizes do not solve the issue: even after resizing twice, the *worse* run keeps producing many duplicates. Interestingly, the *worse* run shows duplicate node creations at the very first garbage collection cycle. Note that these are actual creations of new unique nodes, not requests to create already existing nodes, in case of sharing in the BDDs. Since it is not possible for duplicates to have been created here, as nothing has been removed from the table yet, we can attribute these duplicates to hash collisions. As mentioned before, it remains future work to investigate these hash collision further.

Similar to the duplicate node creations, we can also visualize the duplicate cache entries. Figures 12 and 13 show these duplicate computations. Again, the blue bars represent the fraction of unique computations and the orange bars represent the fraction of duplicate computations. Here, the difference is not as striking as for the node creations. Even the *good* settings produces many duplicate computations, due to the lossiness of the operation cache. However, the *good* settings seem to perform more unique computations than the *worse* settings, as the former peaks around a fraction of 0.4 unique entries for most of the run, while the latter peaks

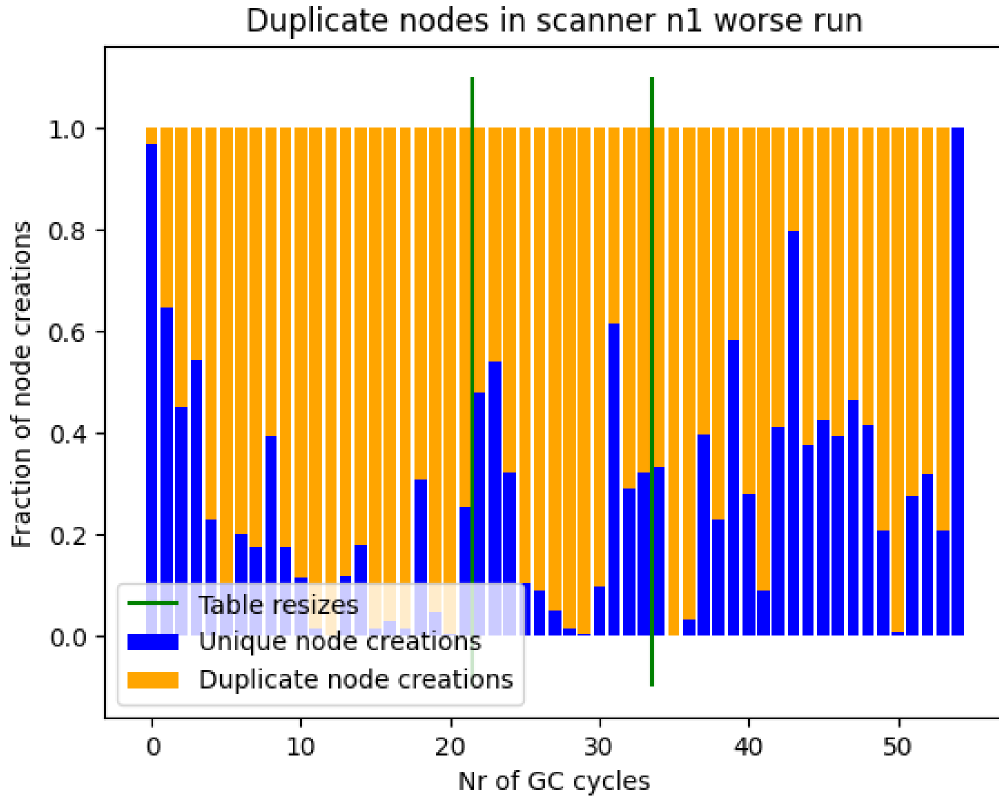


Figure 11: Duplicate node creations during a run of the synthesis algorithm for the `wafer scanner n1` model, with *worse* settings.

around 0.2.

We can also consider the operations for which duplicate cache entries are created. Figures 14 and 15 show the distributions of these duplicate cache entries. The abbreviations used for the operations are shown in Table 3. Clearly, most of the duplicate operations are `relnextIntersection` operations, followed by `relprevIntersection` operations. A possible explanation for this is that a call to the saturation algorithm performs more calls to these operations than recursive calls to itself, resulting in more cache entries for these operations than for the saturation operation. These relational product cache entries then overwrite existing cache entries, resulting in more duplicate work that needs to be performed. This already duplicate work then does the same, overwriting even more cache entries, resulting in many duplicate `relnextIntersection` and `relprevIntersection` operations.

Abbreviation	**Operation**
rn	`relnext`
rni	`relnextIntersection`
rnu	`relnextUnion`
rp	`relprev`
rpi	`relprevIntersection`
rpu	`relprevUnion`
sb	`saturationBackward`
sf	`saturationForward`
bsb	`boundedSaturationBackward`
bsf	`boundedSaturationForward`

Table 3: The abbreviations used for relational product operations, as well as saturation operations.

Step 3: Overview of data

The data shows a clear difference between good and bad runs. A good run produces few duplicate nodes, while a bad run produces many duplicate nodes. In fact, in some cases the bad runs contain garbage collection cycles

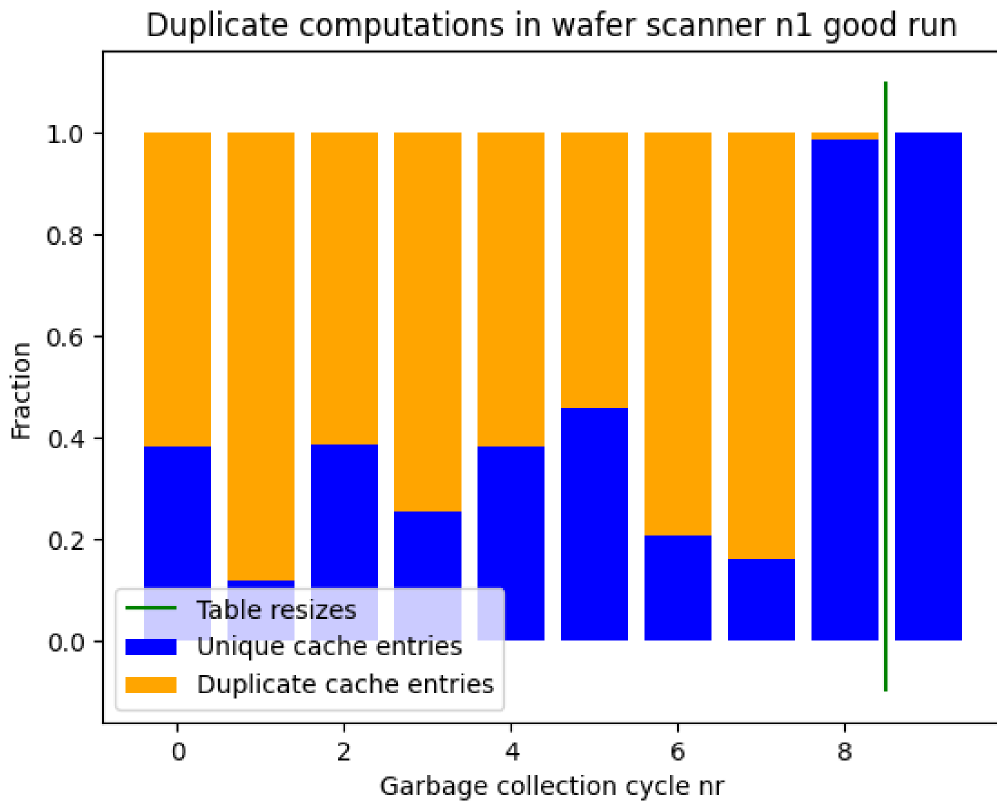


Figure 12: Duplicate computations during a run of the synthesis algorithm for the `wafer scanner n1` model, with *good* settings.

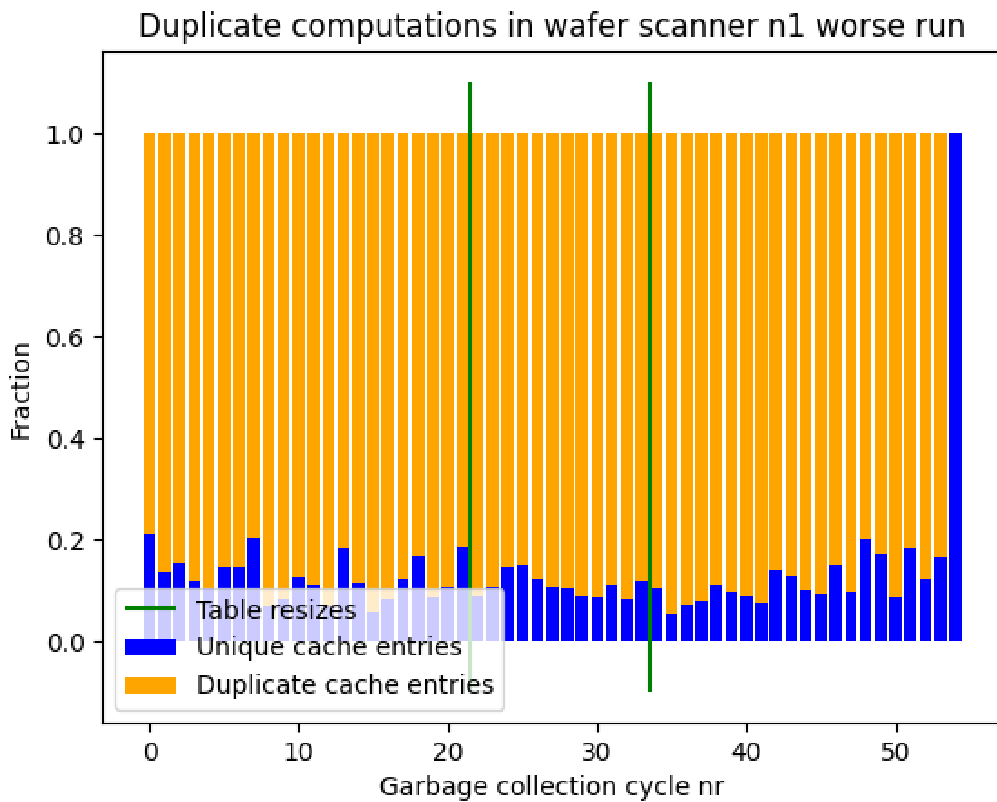


Figure 13: Duplicate computations during a run of the synthesis algorithm for the `wafer scanner n1` model, with *worse* settings.

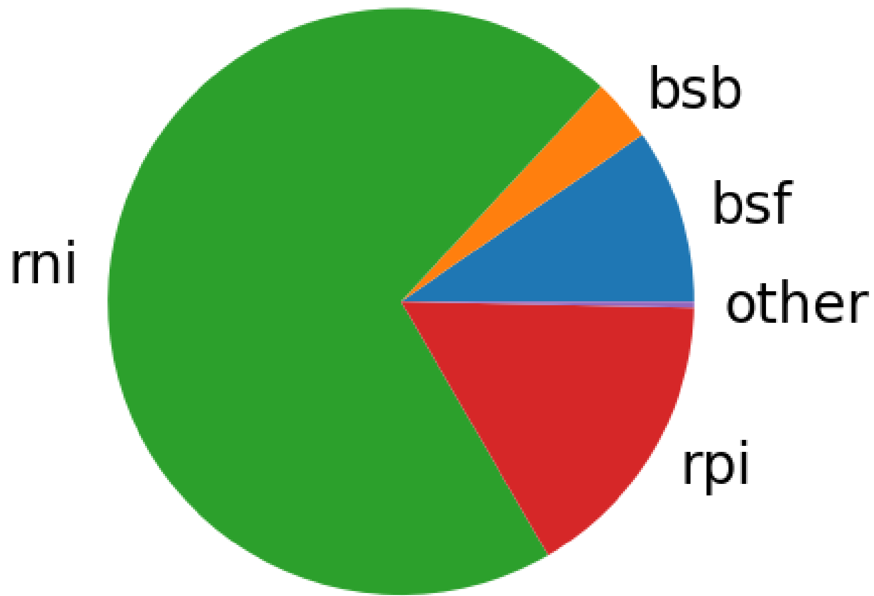


Figure 14: The distribution of duplicate operations for the `wafer scanner n1` model, with *good* settings.

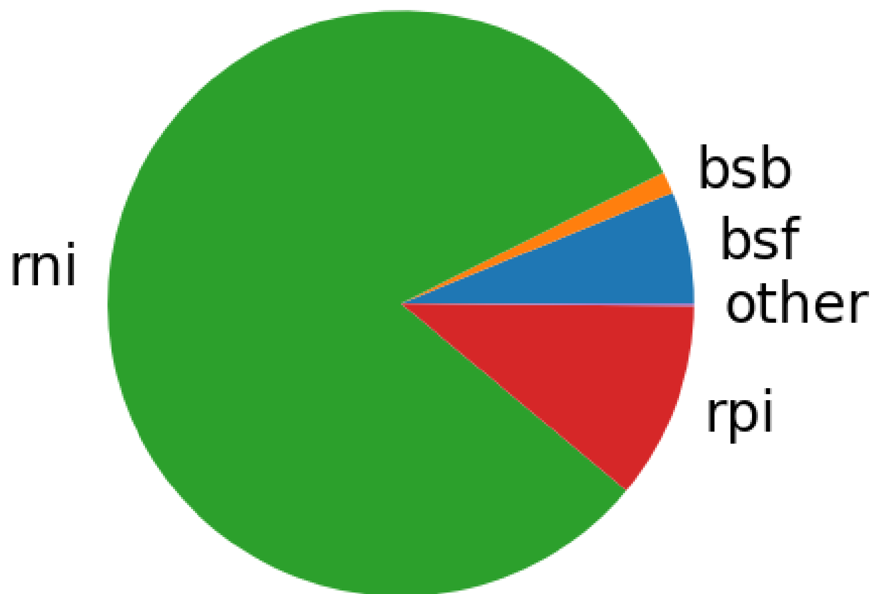


Figure 15: The distribution of duplicate operations for the `wafer scanner n1` model, with *worse* settings.

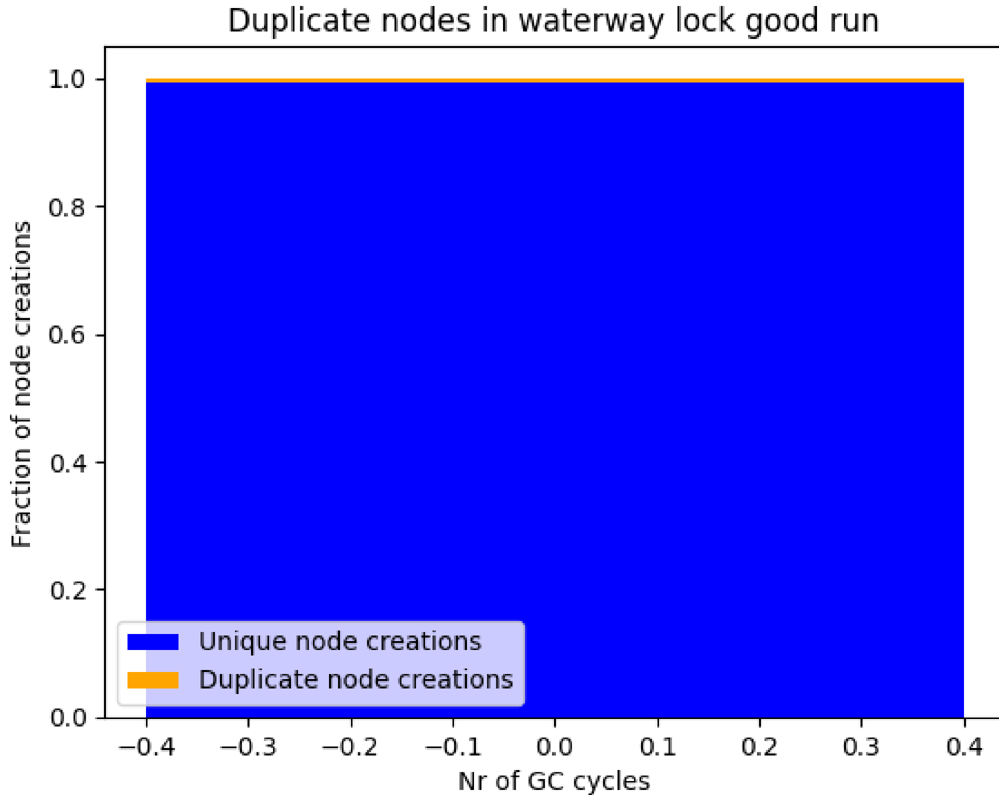


Figure 16: Duplicate node creations during a run of the synthesis algorithm for the `waterway lock` model, with *good* settings.

where only duplicate nodes are produced. When it comes to duplicate cache entries, the difference becomes less obvious, but still seems to be there. Therefore, this perspective still seems promising enough to continue with.

Step 4: Differentiating good and bad runs

As seen in Figure 10 and Figure 11, there is a clear distinction between good and bad runs. In fact, Figure 10 shows mostly blue bars, indicating many unique node creations, while Figure 11 shows large orange areas, where mostly duplicate node creations occurred. This means that these duplicate node creations can indeed be used to identify good and bad runs. This means that at least at the end of a run, these duplicate node creations can be used to distinguish between good and bad runs.

To double-check this, we perform the same measurements for the `waterway lock` model. This time, we take the *good* and *bad* settings, as the difference between *good* and *bad* is less than the difference between *good* and *worse*. Again, Figure 16 and Figure 17 tell the same story. In fact, the *good* synthesis run does all of its work in a single garbage collection cycle, with no duplicate nodes. The *bad* run needs almost 100 garbage collection cycles, and creates mostly duplicate nodes. Figure 18 and Figure 19 show that the same holds for the duplicate computations. The *good* run performs few duplicate computations, while the *bad* run performs many of them. The distribution of duplicate computations for the *good* settings is rather evenly spread, as seen in Figure 20, while for the *bad* settings it is dominated by `relprevIntersection` operations, as seen in Figure 21.

Step 5: Differentiating good and bad runs early

From the data seen previously, it seems that the method of measuring duplicates is also suitable for detecting bad runs early. This could be due to the fact that when a run gets stuck creating many duplicates, it cannot easily get unstuck. When it performs duplicate computations, it overwrites necessary cache entries. In the meantime, the unique table fills up with intermediate nodes. Upon garbage collecting, obsolete intermediate nodes are removed. From the data, it shows that the unique table does not resize quickly, so that the cache table does not resize either. This way, more and more useful cache entries are overwritten, so that the run does not get unstuck.

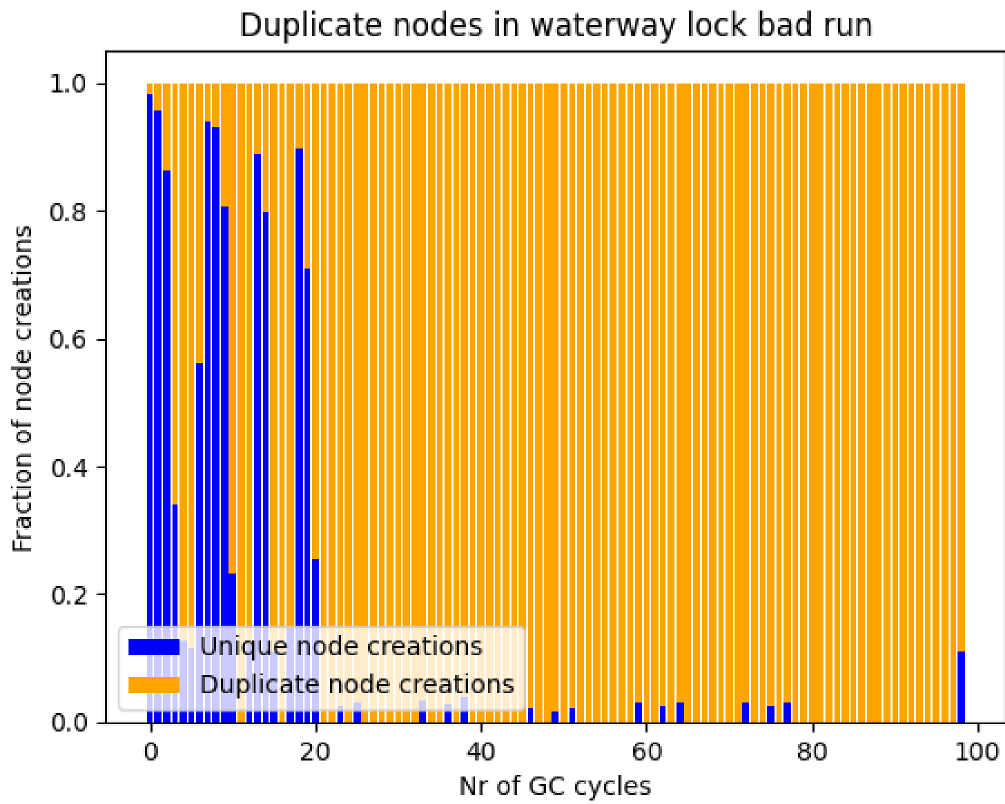


Figure 17: Duplicate node creations during a run of the synthesis algorithm for the `waterway lock` model, with *bad* settings.

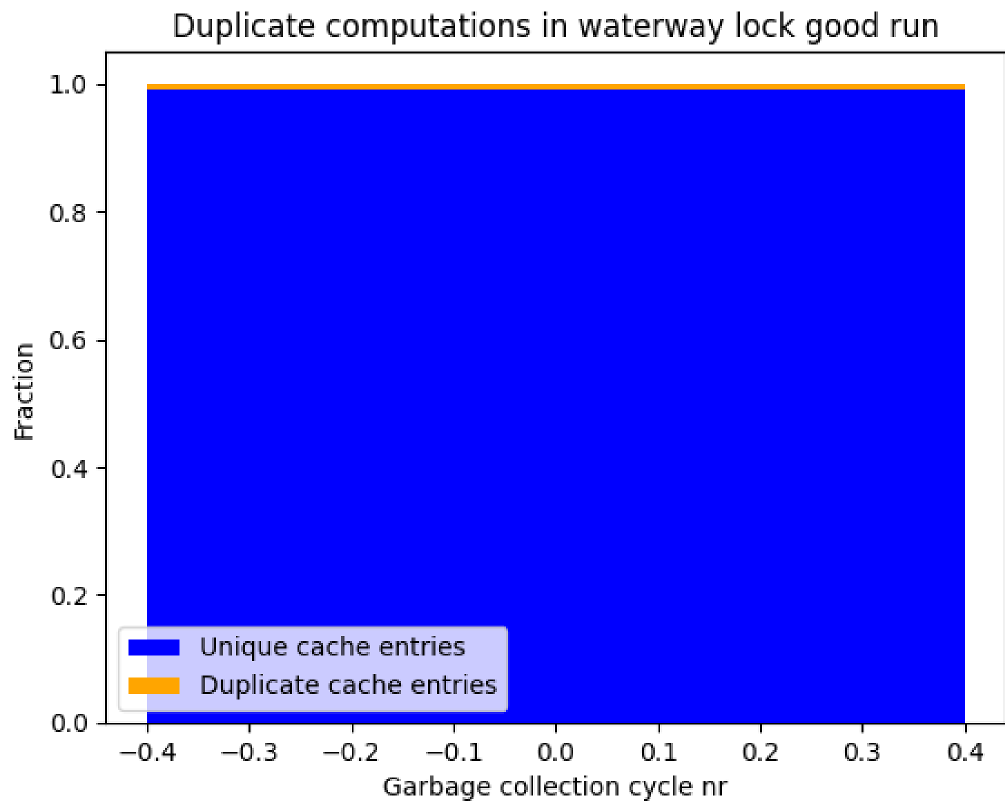


Figure 18: Duplicate computations during a run of the synthesis algorithm for the `waterway lock` model, with *good* settings.

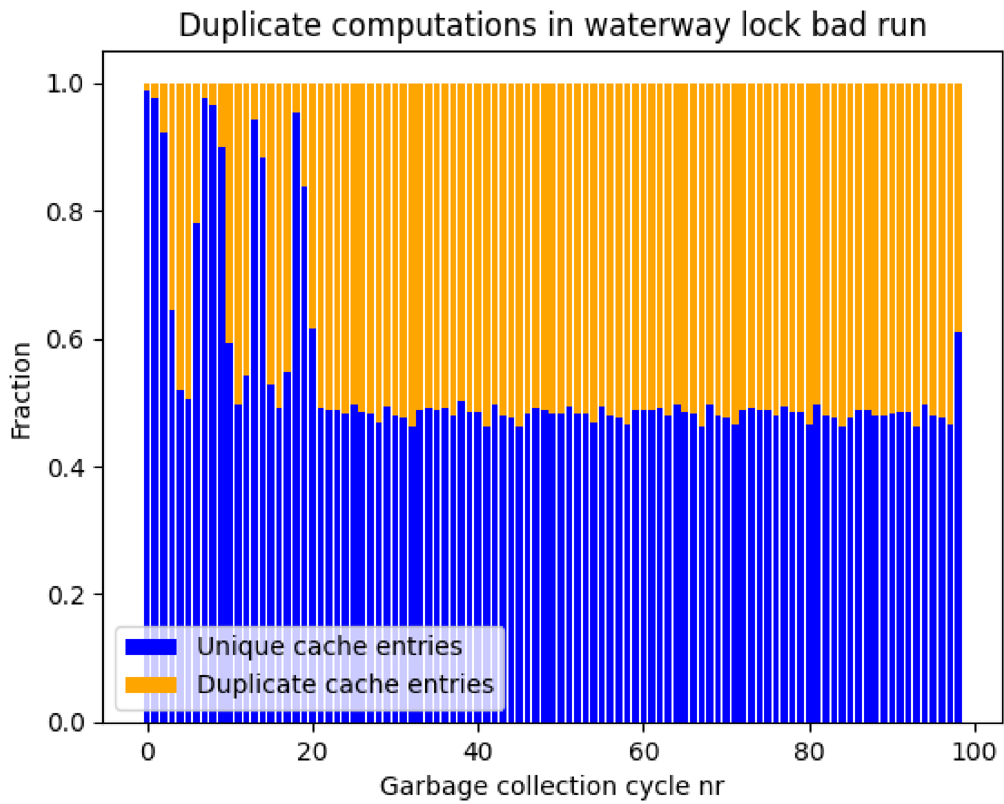


Figure 19: Duplicate computations during a run of the synthesis algorithm for the `waterway lock` model, with *bad* settings.

Figure 20: The distribution of duplicate operations for the `waterway lock` model, with *good* settings.

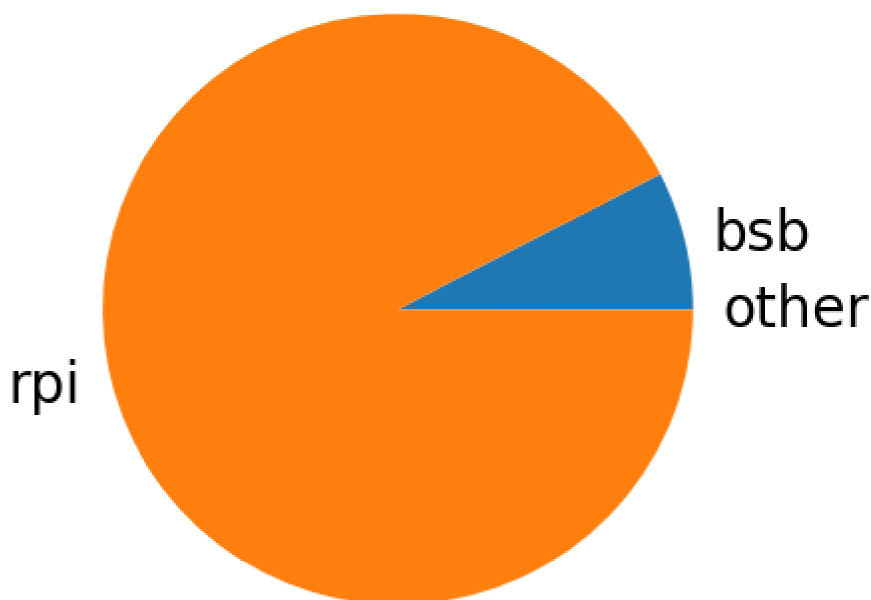


Figure 21: The distribution of duplicate operations for the `waterway lock` model, with *bad* settings.

Step 6: Predictive power

Now that it seems that the measurement of duplicates seems to be able to differentiate good and bad runs early, we can consider whether it is worth investigating further, and whether it can be made actionable. Since the data is easy to collect, and the difference in this data between good and bad runs seems enormous, this perspective is a good candidate to be considered for making a predictor in Section 5.

4.6 Evaluation

Now that we have explored the saturation algorithm from three perspectives, we can make an informed decision on what could be an indicator for good and bad runs. The next step will then be to construct a predictor that can be used to recognize when a run has gone bad. What follows is a short summary of the conclusions reached for each of the three perspectives. We end this section by selecting the most promising perspective, to turn into a predictor in the next section.

Through the input perspective, we explored what happens to the input states BDD when it is manipulated by the saturation algorithm. We visually animated and explored the evolution of this states BDD throughout the saturation algorithm. Here, we noticed that some bad runs seem to show a blow up, where many BDD nodes are created quickly, spread all over the visualization. However, when further exploring this idea, we did not find a very strong correlation between this blow up and a bad run. Therefore, we made the decision to focus on the other perspectives.

Using the inner workings perspective, we studied the order in which the saturation algorithm applied its transition relations. Here, we introduced the idea of the velocity of the saturation algorithm. At first it seemed that this velocity could be used to differentiate between good and bad runs. However, when further studying this idea, it became clear that this idea also did not show much correlation between a certain velocity and a bad run. Therefore it was discontinued in favor of the data structure perspective.

With the data structure perspective, we investigated how the unique table and cache table data structures in JavaBDD affected the performance of the saturation algorithm. By storing all previously created BDD nodes and cache entries, regardless of garbage collections and cache entry overwrites, we find that the saturation algorithm creates many duplicate BDD nodes, which results in many duplicate computations. Here, the difference between a good and bad run is clear, as bad runs create many more duplicates than good runs. Therefore, these duplicates are a promising indicator for differentiating between good and bad runs.

Having explored all three perspectives, we can conclude that the duplicate node creations and duplicate

cache entries are the most promising indicators for a bad run. In the next section, we explore ways to turn these indicators into a predictor that can determine whether a run is good or bad.

5 Slowdown predictors

Now that we have identified duplicate node and cache entry creations as possible indicators for bad runs, we would like to make this data actionable. Therefore, we want to construct a *predictor* that predicts whether a given run is bad. To be the most useful, this predictor should indicate that a run is bad as early in the run as possible. Additionally, to not disturb the user too much, the predictor should have little runtime overhead in terms of runtime and memory usage. Again, all experiments performed in this thesis can be found in the artifact that accompanies this thesis [23].

5.1 Node duplicates versus cache entry duplicates

First, we need to determine which data to use for constructing predictors. We have the choice between the duplicate node creation data and the duplicate cache entry data. As seen in section 4.5, both can be used as an indicator for bad runs. Even though the duplicate node creations seem more indicative, we can still look at the practical side of collecting this data, by measuring the cost of keeping track of these variables. Ideally, we will then build a predictor out of the variable that is cheapest to keep track of. To determine which one that is, we measure the time and memory consumption of selected synthesis runs. Figure 22 shows timing measurements for the *good* runs of the `waterway lock` and `wafer scanner n1` models, and Figure 23 shows the same for the *bad* runs. For each timing measurement, 10 synthesis runs were performed, and the average was taken. Each graph shows three configurations: one without any additional measurements, one with only the duplicate node detection measurements, and one with both duplicate node and duplicate cache entry measurements. Note that we do not perform the cache entry measurements separately, as we still need the unique table adaptations described in Section 4.5. As seen from these figures, the duplicate node detection measurements have a relatively small impact on the performance of synthesis, whereas the duplicate cache entry measurements have considerable impact. We have also observed that the duplicate cache entry measurements have considerable impact on the memory usage of the application, but since memory measurements in Java are tricky, we leave this as future work. Taking this into account, we use the measurements on duplicate nodes to construct the predictor with, and disregard the duplicate cache entries.

5.2 Slowdown predictors

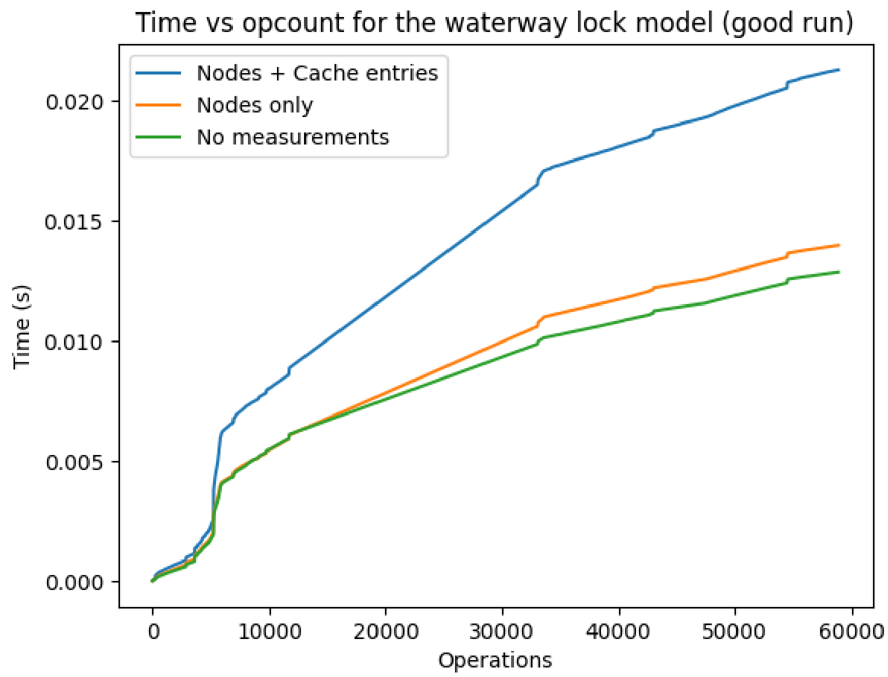
Now that we have decided to use only the duplicate node detection measurements, we would like to find a symptom for determining when a synthesis run is bad. To do this, we use the concept of *slowdown predictors*. These predictors are classifiers that take the duplicate node creation data of a run, look for a certain symptom in that data, and use it to determine whether a run is bad. In order to come up with these predictors, we need to determine what constitutes a good predictor. Most importantly, it should flag a run as bad as early as possible. It makes no sense for the user to have to wait until the run is nearly finished, before it finally flags it as bad. As a consequence, it may be inevitable that it flags some good runs as bad too. Something could be said about having the predictor be able to change its answer from bad to good. When it detects that fewer duplicates are being created, it may decide to switch back to classifying the run as good. Lastly, a good predictor should be simple, so that the user can understand immediately why a run was given a certain label.

Table 4 lists the predictors that are studied in this section. All of these predictors are based on the duplicate node data. They all work during a run, instead of just at the end of a run. By default, they classify a run as good, unless they find some symptom of a bad run. The *TotalDuplicatesPredictor* looks at the entire run so far. If the fraction of duplicates compared to unique creations exceeds a threshold, it flags the run as bad. The *NCyclesPredictor* works similarly, but only considers the last N garbage collection cycles to mark a run as bad. The *NoProgressPredictor* classifies a run as bad when there have been a certain amount of garbage collection cycles without any unique node creations. The *VeryLittleProgressPredictor* relaxes this condition, and marks a run as bad when there have been a certain amount of garbage collection cycles with less than 1% unique node creations.

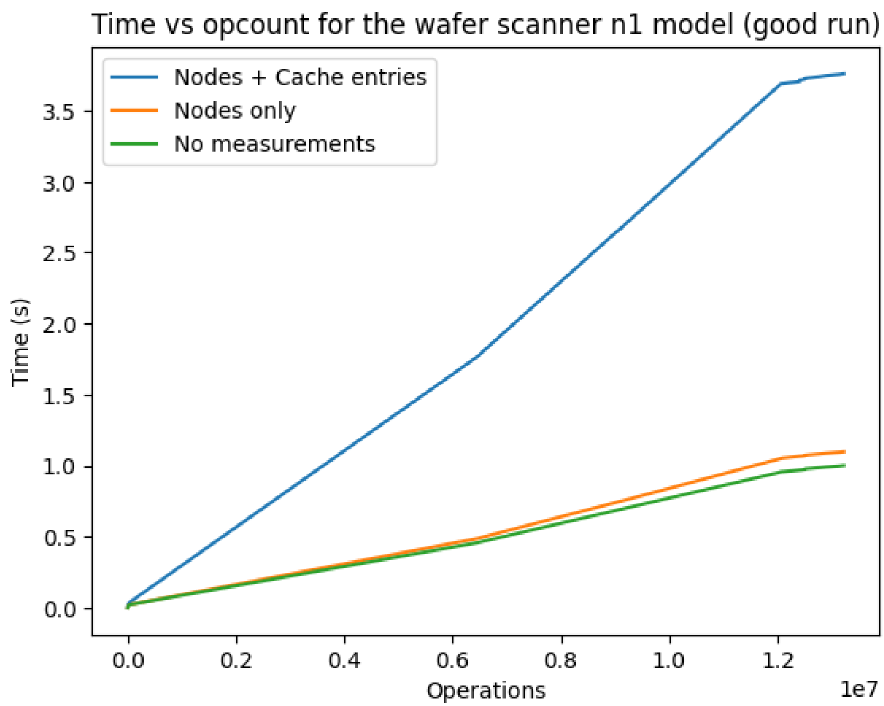
The last three predictors are combinations of other predictors. The *ConjunctionPredictor* looks at the outcome of two other predictors, and classifies a run as bad when both predictors classify it as bad. The *DisjunctionPredictor* also takes two predictors, but classifies a run as bad when at least one of these classifies it as bad. Finally, the *MultiPredictor* takes three predictors, and classifies the run the same as the majority of the input predictors.

5.3 Data collection

All of the predictors listed in Table 4 can be configured using one or more parameters. We would like to find the predictor and associated parameters that is the most accurate at predicting whether a run is good or bad. In order to do this, we need to have data on many runs of different models, so that we can first tune the parameters

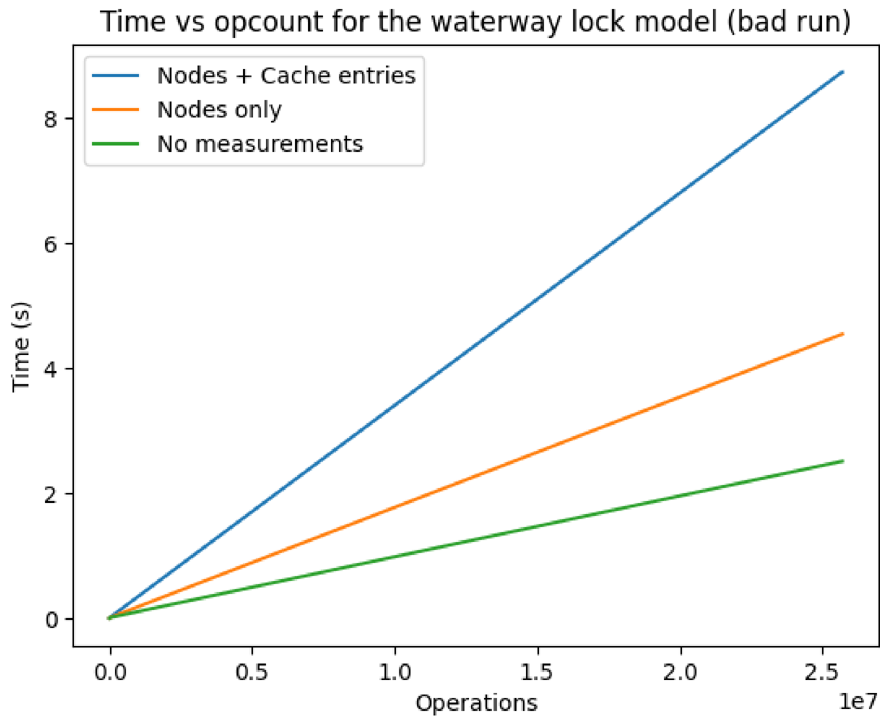


(a) Timing measurements for the *good* runs of the *waterway lock* model.

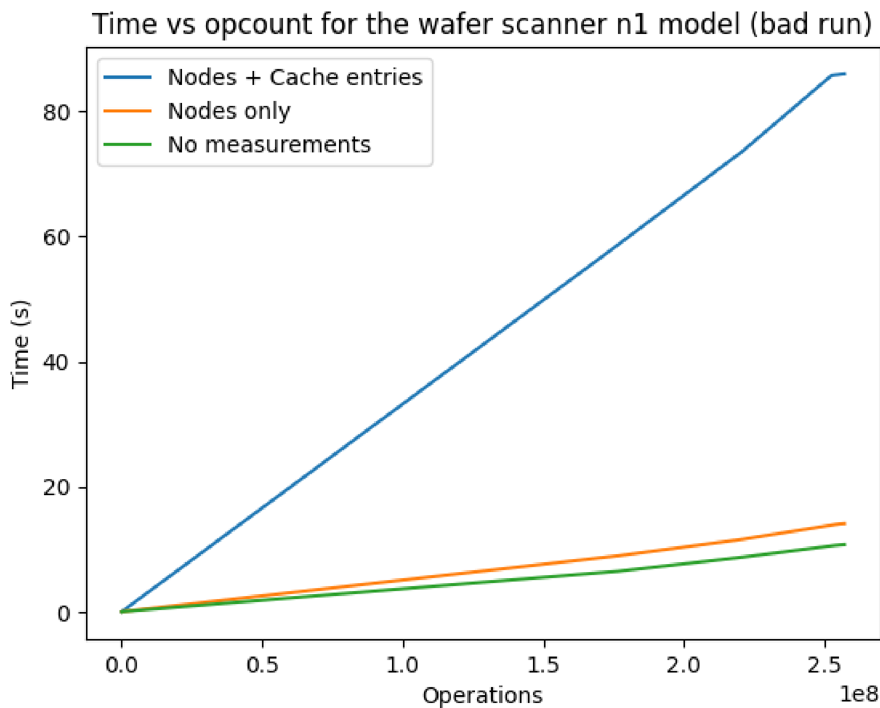


(b) Timing measurements for the *good* runs of the *wafer scanner n1* model.

Figure 22: Timing measurements for *good* runs of the *waterway lock* model and the *wafer scanner n1* model.



(a) Timing measurements for the *bad* runs of the *waterway lock* model.



(b) Timing measurements for the *bad* runs of the *wafer scanner n1* model.

Figure 23: Timing measurements for *bad* runs of the *waterway lock* model and the *wafer scanner n1* model.

Predictor	Description
$TotalDuplicatesPredictor(t)$	Classifies a run as bad when the fraction of duplicates over the entire run exceeds threshold t.
$NCyclesPredictor(n, t)$	Classifies a run as bad when the fraction of duplicates in the last n garbage collection cycles exceeds threshold t. It never classifies a run as bad if the run performs fewer than N garbage collections.
$NoProgressPredictor(n)$	Classifies a run as bad when there have been n garbage collection cycles with only duplicates.
$VeryLittleProgressPredictor(n)$	Classifies a run as bad when there have been n garbage collection cycles with fewer than 1% unique node creations.
$ConjunctionPredictor(p_1, p_2)$	Takes two predictors p_1 and p_2 as arguments, and classifies a run as bad when both predictors classify it as such.
$DisjunctionPredictor(p_1, p_2)$	Takes two predictors p_1 and p_2 as arguments, and classifies a run as bad when at least one of the predictors classifies it as such.
$MultiPredictor(p_1, p_2, p_3)$	Takes three predictors p_1, p_2 and p_3 as arguments, and classifies a run the same as the majority of the three predictors.

Table 4: The different predictors for determining whether a synthesis run is good or bad.

of the predictors, select the best predictor, and then validate the performance of this predictor. Therefore, we need two datasets: one for tuning the predictors, and one for validating the performance of the best predictor.

To create the tuning data, we perform an experiment using version 9.0 of Eclipse ESCET, on version 8.0 of the CIF benchmark models. A detailed description of these benchmark models can be found in [3]. In the experiment, we create a set of 33 variable orders, using different combinations of the available initial orders and ordering heuristics in CIF. The goal of these different variable orders is to create good runs, bad runs, and many runs in between. For each order, we use three different reachability settings: no forward reachability, forward reachability first, and forward reachability last. For clarity, when forward reachability is used, we append `fw` to the name of the model. Since forward reachability adds another computation to the synthesis algorithm, we treat runs with and without forward reachability as separate models. As with the earlier duplicate node measurements, we record the total number of nodes created as well as the number of duplicate nodes created for each garbage collection cycle. Additionally, we measure the time that the algorithm spends performing reachability computations. This produces a dataset with many different runs per model. We create the validation set in the same manner, but use different models instead of the benchmark models. This way, the tuning set and validation set are completely separate. See Section 5.5 for more information on the validation set.

Before we can use this data, we need to label the runs in the data with the *good* and *bad* labels. We have a lot of freedom with this labeling, as the only criterion for a *good* run is that it outperforms *bad* runs. The labeling should be independent of intrinsic model complexity, as good and bad should be relative to the same model. Since the user cares about the runtime of the algorithm, we should label each run according to the time it takes to perform. There is still a small issue with this, however, as we have only performed a single timing measurement for each run, which means that there could be a lot of variance in the measurements. Since it takes many hours to perform the several thousand runs, especially when every run has to be performed multiple times, we decide to instead use another technique to solve this. We construct a model for the runtime t from the collected timing data, such that the variances of the individual runs average out. We model the time t as a linear combination of the number of operations and the number of garbage collections performed. This is a refinement of the work by [24], where the runtime is modeled as a linear function of the number of BDD operations. However, we have many runs where many garbage collections are performed, so that the time spent

Table 5: Overview of the benchmark models used. Adapted from [3].

Nr	Short name	Full name	Refs	Size (US)	Size (CS)
1	adas	Advanced driver assistance system	[25]	$3.40 \cdot 10^{09}$	$2.04 \cdot 10^{10}$
2	bcs-static	Body comfort system (static)	[26, 27]	$3.18 \cdot 10^{14}$	$1.97 \cdot 10^{14}$
3	bridge	Bridge	[28]	$5.92 \cdot 10^{33}$	$9.55 \cdot 10^{28}$
4	cluster-tool	Cluster tool	[29]	$7.50 \cdot 10^{06}$	$2.70 \cdot 10^{06}$
5	festo	FESTO production line	[30]	$1.47 \cdot 10^{26}$	$2.22 \cdot 10^{25}$
6	litho-init	Lithography machine initialization	[31, 32]	$7.21 \cdot 10^{09}$	$2.46 \cdot 10^{09}$
7	mri-pss-event	MRI scanner PSS (event-based)	[33, 34]	$3.60 \cdot 10^{03}$	$3.09 \cdot 10^{04}$
8	mri-pss-state	MRI scanner PSS (state-based)	[34]	$1.44 \cdot 10^{04}$	$2.79 \cdot 10^{04}$
9	multi-agent-form	Multi-agent formation	[35]	$1.00 \cdot 10^{03}$	$3.04 \cdot 10^{02}$
10	prod-cell	Production cell	[36]	$3.76 \cdot 10^{08}$	$1.15 \cdot 10^{08}$
11	theme-park	Theme park vehicles	[37]	$2.95 \cdot 10^{05}$	$6.69 \cdot 10^{06}$
12	wafer-scanner-n1	Wafer scanner ($n=1$)	[38]	$5.30 \cdot 10^{05}$	$5.24 \cdot 10^{04}$
13	waterway-lock	Waterway lock	[39]	$5.96 \cdot 10^{32}$	$5.87 \cdot 10^{24}$

garbage collecting cannot be overlooked. The model we use here is $t = t_{gc}g + t_o o$, where g is the number of garbage collections performed, t_{gc} the time it takes to perform a garbage collection, o the number of BDD operations performed and t_o the time it takes to perform a BDD operation. Then, to find t_{gc} and t_o, we can fit this model to the collected data. This results in values of $t_{gc} = 5.0ms/gc$ and $t_o = 1.6 \cdot 10^{-4}ms/op$.

Now that we have a model for the runtime of the algorithm, we can compute the runtime for each run by plugging the number of garbage collections and operations performed into the model. Now we can actually label the data based on the run that has the lowest runtime t_{lowest}. When a run has a runtime higher than $3t_{lowest}$, we label it as *bad*. Otherwise, we label it as *good*. The factor 3 is chosen rather arbitrarily, but seems reasonable as a user does not want to wait three times longer than necessary.

The dataset now contains both good and bad runs for many models. However, for some of the smaller models no bad runs were produced. Therefore, we perform the experiment again, now adding data obtained by changing the BDD initial node table to only 5,000 nodes, instead of the default value of 100,000 nodes. For some models, this does in fact produce bad runs, while for others it still is not enough. We decide to include only models that have both good and bad runs in the dataset in the tuning and validation of the predictors. Table 5 shows an overview of the models that produce both good and bad runs, and therefore are used in the experiments. The exact distribution of good and bad runs differs per model, where some models have more good than bad runs, and vice versa. In total, the dataset contains 1859 good runs and 1061 bad runs. For some models only the runs with forward reachability are used, as the ones without forward reachability did not result in any bad runs being created.

5.4 Predictor tuning

Now that we have a dataset with many runs on many models, we can look for a slowdown predictor. As seen in Table 4, there are quite a few predictors, and all of them have parameters that can be tuned. We want to select the predictor that works best and tune it to get the best results. To do this, we apply many predictors with different settings to the data, and select the best one. To simulate how a predictor acts during a run, we consider the first parts of each run, and see whether the predictor can correctly classify the run at that stage. We divide the runs into increasingly larger sections, where the first section contains the first 10% of the run, the second section contains the first 20% of the run and so forth, all the way up to 100%.

Figures 24, 25, 26 and 27 show the results for each of the four basic predictor types. Since the *NCyclesPredictor* has two parameters instead of one, it is shown in a 3×3 grid instead of in a single plot. What follows is an interpretation of each of these figures.

Figure 24 shows an overview of the performance of different configurations of the *TotalDuplicatesPredictor*. The figure shows a plot divided into cells. Each cell represents the average score of a *TotalDuplicatesPredictor*, after having seen a certain part of a run. The x-axis of the plot shows the different thresholds used for the predictor. The y-axis, read from top to bottom, shows what percentage of the run is considered. Each cell then represents a score, which is indicated by its color. The bar on the right of the plot shows what score is represented by which color. The score of a given cell with threshold t and percentage p is calculated as follows: for each model, have the predictor with threshold t predict every run based on the first p percentage of the run. Then find the fraction of correct predictions for this model. Lastly, find the final score by averaging over all models. Each cell thus represents an average accuracy over all models.

The figure shows that a threshold of 0.0 performs awfully. With this threshold, it flags every run that has a duplicate node creation as bad. Since duplicate node creations occur even for good runs, the predictor misclassifies these runs. On the other end of a spectrum, a threshold of 1.0 is equally bad. This predictor flags a

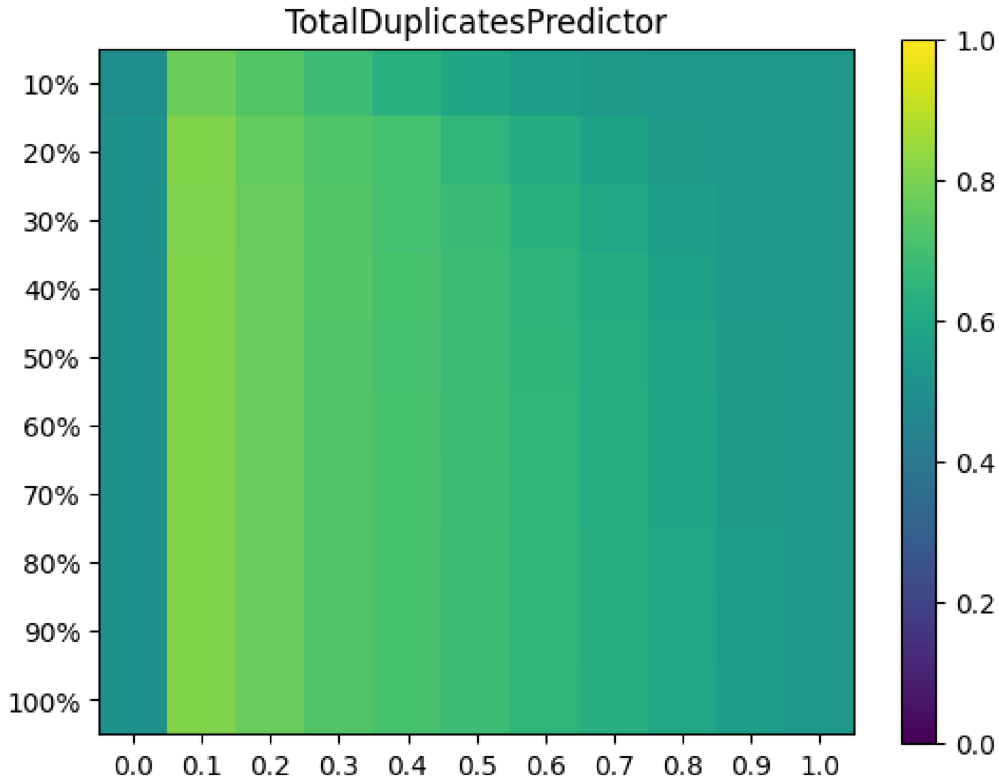


Figure 24: An overview of the performance of the *TotalDuplicatesPredictor*, with thresholds ranging from 0.0 to 1.0. The plot shows that a threshold of 0.1 performs best.

run as bad when it consists of only duplicates. As seen by the brightness of the columns, the performance seems to peek with a threshold of 0.1. Larger thresholds are too strict, as indicated by the decreasing brightness. Therefore, we will consider this threshold for further evaluation.

Figure 25 shows an overview of the performance of different configurations of the *NCyclesPredictor*. The figure shows a 3×3 grid of plots for the predictors, corresponding to window sizes 1 through 9. Again, the x-axis of each plot corresponds to the threshold used. In general, the plots shown are a bright green color, which indicates that the predictor performs well. As seen in the plot, the bigger the window size, the worse the predictor gets. Apparently the predictor is too strict when the window is too large. When zooming in on a single window size, it becomes clear that setting the threshold too high is also detrimental to the predictor performance. There are three configurations that stand out: a window size of 1 with a threshold of 0.2, a window size of 2 with a threshold of 0.1 and a window size of 3 with a threshold of 0.1. We will keep these three configurations in mind for further investigation.

Figure 26 shows an overview of the performance of different configurations of the *NoProgressPredictor*. Compared to the previous two predictors, this predictor performs poorly. A threshold of 1 performs best, but still does not compare to the *TotalDuplicatesPredictor* or the *NCyclesPredictor*. Evidently counting cycles without unique node creations is not a good strategy for distinguishing good and bad runs. This is probably due to the fact that not many bad runs have cycles without any unique node creations. Since it is clearly outperformed by the *TotalDuplicatesPredictor* and *NCyclesPredictor*, we decide to not continue with this predictor.

Figure 27 shows an overview of the performance of different configurations of the *VeryLittleProgressPredictor*. It performs similarly to the *NoProgressPredictor*. Only when examining the average performance more closely do we see that the *VeryLittleProgressPredictor* slightly outperforms the *NoProgressPredictor*, by a few percent at every fraction of the run. However, in total, both predictors do not reach more than 68% accuracy after having seen an entire run. Therefore, we also decide to leave this predictor out of consideration.

As seen in the plots, the *TotalDuplicatesPredictor* and *NCyclesPredictor* show the best results. We can compare some of the best configurations in order to find out which predictor works best. For the *TotalDuplicatesPredictor* we consider the predictor with threshold 0.1. For the *NCyclesPredictor* we consider the configurations $(1, 0.2)$, $(2, 0.1)$ and $(3, 0.1)$, as these seem to be the most promising. When comparing predictors, we are interested in what kind of misclassifications they make. In our case, a *false positive* occurs when the predictor labels a good run as bad, and a *false negative* occurs when the predictor labels a bad run as good. Since we do not want to bother the user when it is not necessary, we would like our predictor to have as few false positives as possible, while still having a good average score. It is better to have a bad run continue a little longer than

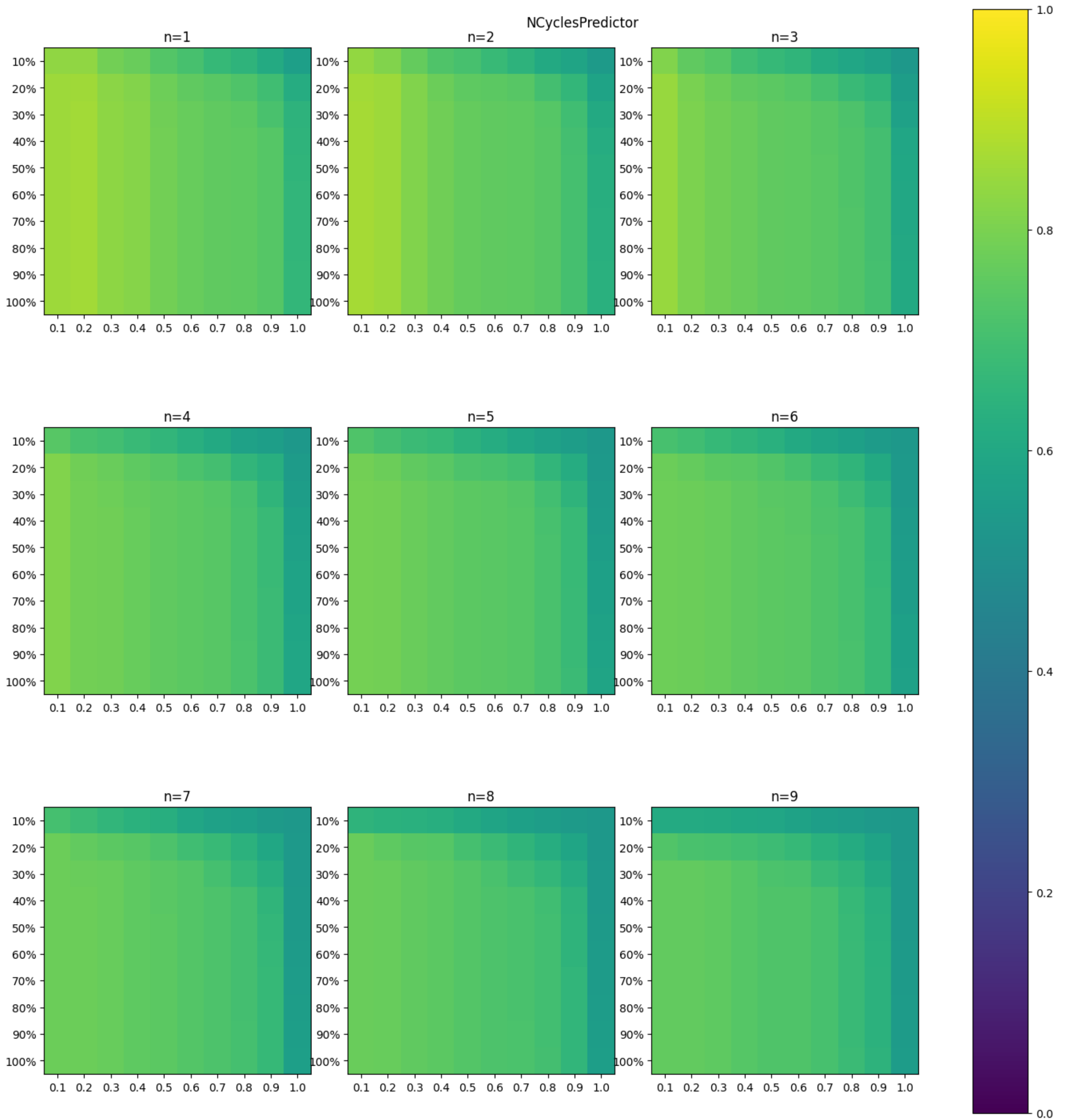


Figure 25: An overview of the performance of the *NCyclesPredictor*, with window sizes ranging from $n = 1$ to $n = 9$, and thresholds ranging from 0.1 to 1.0. The settings that stand out here are $(1, 0.2)$, $(2, 0.1)$ and $(3, 0.1)$.

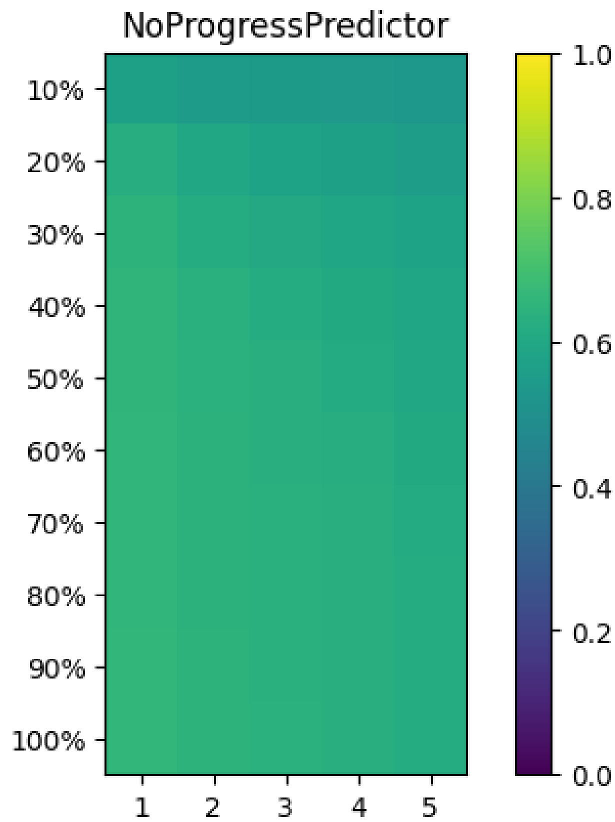


Figure 26: An overview of the performance of the *NoProgressPredictor*, with thresholds ranging from 1 to 5. A threshold of 1 performs best, but overall the performance of the predictor is weak.

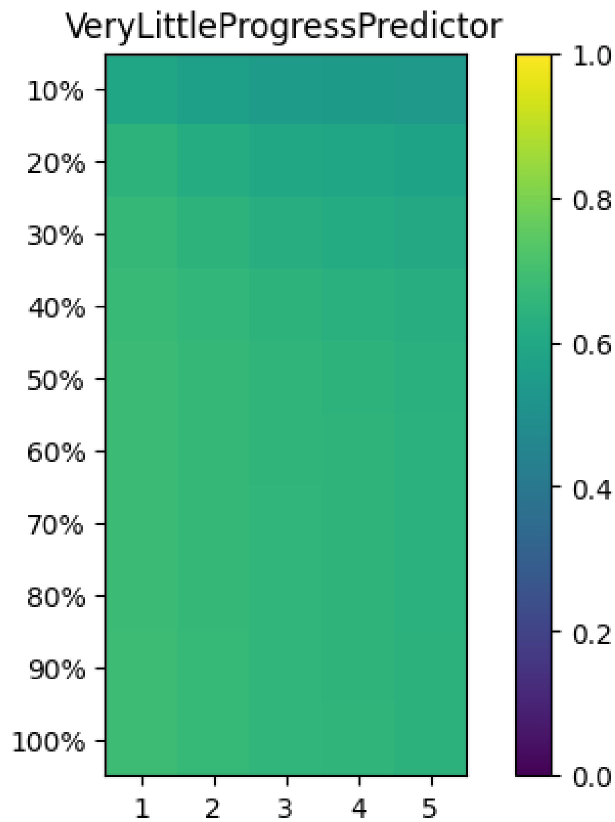


Figure 27: An overview of the performance of the *VeryLittleProgressPredictor*, with thresholds ranging from 1 to 5. A threshold of 1 performs best. This predictor outperforms the similar *NoProgressPredictor*, but still performs weakly compared to the other predictors.

Predictor	*Average (30%)*	*Good runs (30%)*	*Bad runs (30%)*
NCyclesPredictor(1,0.2)	0.86	0.87	0.85
NCyclesPredictor(2,0.1)	0.87	0.85	0.88
NCyclesPredictor(3,0.1)	0.85	0.89	0.80
TotalDuplicatesPredictor(0.1)	0.81	0.80	0.82

Table 6: Comparison of the accuracies of several predictors, after having seen 30% of a run.

Predictor	*Average (30%)*	*Good runs (30%)*	*Bad runs (30%)*
ConjunctionPredictor	0.84	0.92	0.76
DisjunctionPredictor	0.81	0.77	0.86
MultiPredictor	0.84	0.92	0.76

Table 7: Comparison of the accuracies of several combinations of predictors, after having seen 30% of a run.

to have the user terminate a good run early. Table 6 shows a comparison of the five aforementioned predictor configurations, after having seen 30% of a run. This threshold of 30% was chosen as the average score of the predictors seem to remain the same after having seen 30% of a run. This allows us to compare predictors. The table shows that the three *NCyclesPredictors* have similar scores on average. The *NCyclesPredictor(1,0.2)* can accurately predict 86% of runs after having seen 30% of the run, the *NCyclesPredictor* can predict 87% of runs correctly, and the *NCyclesPredictor(3,0.1)* has an accuracy of 85%. In order to make a decision on which one of these is the best, we should consider how many false positives and false negatives it produces. As seen in the *good runs* column, the *NCyclesPredictor(1,0.2)* correctly predicts 87% of good runs, meaning it produces false positives in 13% of runs. The *NCyclesPredictor(2,0.1)* performs slightly worse in this regard, producing false positives in 15% of the runs. The *NCyclesPredictor(3,0.1)* only produces false positives in 11% of runs. When it comes to false negatives, the *NCyclesPredictor(1,0.2)* produces false negatives in 15% of cases. The *NCyclesPredictor(2,0.1)* produces false negatives in 12% of cases, and the *NCyclesPredictor(3,0.1)* produces false negatives in 20% of cases. Since we have decided that false positives are worse than false negatives, and all three predictors have similar average scores, we decide that the predictor with a window size of 3 and a threshold of 0.1 performs best. For the *TotalDuplicatesPredictor*, the threshold of 0.1 works best. It is still outperformed by the *NCyclesPredictor*, however. Still, it may be useful to consider it when combining predictors. From now on, when we refer to the *TotalDuplicatesPredictor* we assume it has a threshold of 0.1, and when we refer to the *NCyclesPredictor* we mean the configuration with a window size of 3 and a threshold of 0.1.

Now that we have identified the *NCyclesPredictor* as the best predictor, we can examine how well it performs exactly. To do this, we can plot the results of this predictor in more detail, by showing its performance on every individual model separately. Figure 28 shows the overall results for this predictor. Across the board, it seems to perform well, as it is able to classify many runs correctly early on. However, it seems to struggle with classifying runs of the `bcs dynamic fw`, `bridge fw` and `mri event fw` models. To see what is going on, we can plot the good and bad runs separately. Figures 29 and 30 show the performance of the predictor on the good and bad runs respectively. As seen in Figure 29, the predictor produces many false positives on the `bridge fw` model. Looking into the data of this model, it seems that even the best run in the dataset produces many duplicate nodes. Since the predictor uses only these duplicate nodes to base its prediction on, it classifies all of these runs as bad. This could mean that the model inherently produces many duplicates, or that we have not been able to find a proper good run, where few duplicates are created. Also, good runs of the `mri event` and `wafer scanner` models seem to be difficult to classify. Again, runs of these models produce many duplicate nodes. Future work is needed to know whether this can be avoided, or that it is inherent to the model.

Looking at Figure 30, we see that the predictor struggles with bad runs of the `bcs dynamic fw`, `mri event fw` and `waterway lock fw` models. Interestingly, all of these models have the forward reachability computation setting enabled. Closer inspection of the data generated by these three models shows that the misclassified bad runs in these models do not generate many duplicate nodes. However, since the runs are labeled as bad, they still take much longer than the best run for these models. This could indicate that some other factor causes these runs to perform poorly. It remains future work to determine which factor this could be, and how the settings influence this factor.

Now that we have identified which configurations work best, we can attempt to combine predictors using the *ConjunctionPredictor*, *DisjunctionPredictor* and *MultiPredictor*. Here, we can try to combine the *TotalDuplicatesPredictor* with the *NCyclesPredictor*, in an attempt to have them reduce each others weaknesses. Since the *MultiPredictor* takes three predictors, we add the *VeryLittleProgressPredictor(1)* as third argument, so that it consists of three different predictors. Table 7 shows the results of these combinations of predictors.

First, we analyze the *ConjunctionPredictor*, which is the conjunction of the *TotalDuplicatesPredictor* and the *NCyclesPredictor*. This new predictor is more strict when it comes to labeling a run as bad, since it requires

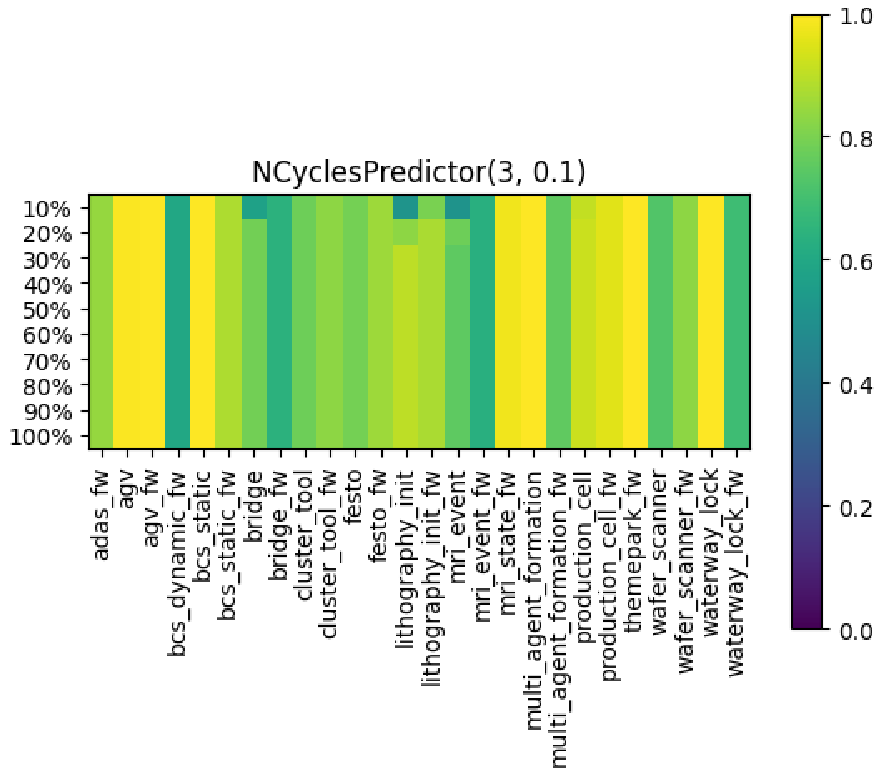


Figure 28: Plot of the performance of the *NCyclesPredictor* using a window size of 3 cycles, with the threshold set to 0.1.

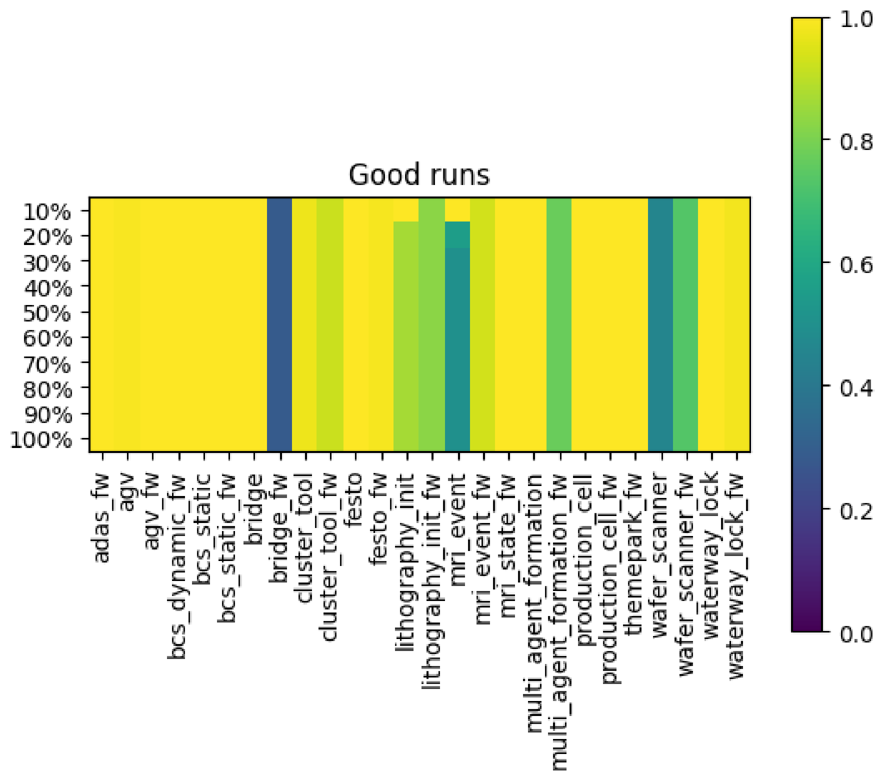


Figure 29: Plot of the performance of the *good* runs of the *NCyclesPredictor* using a window size of 3 cycles, with the threshold set to 0.1.

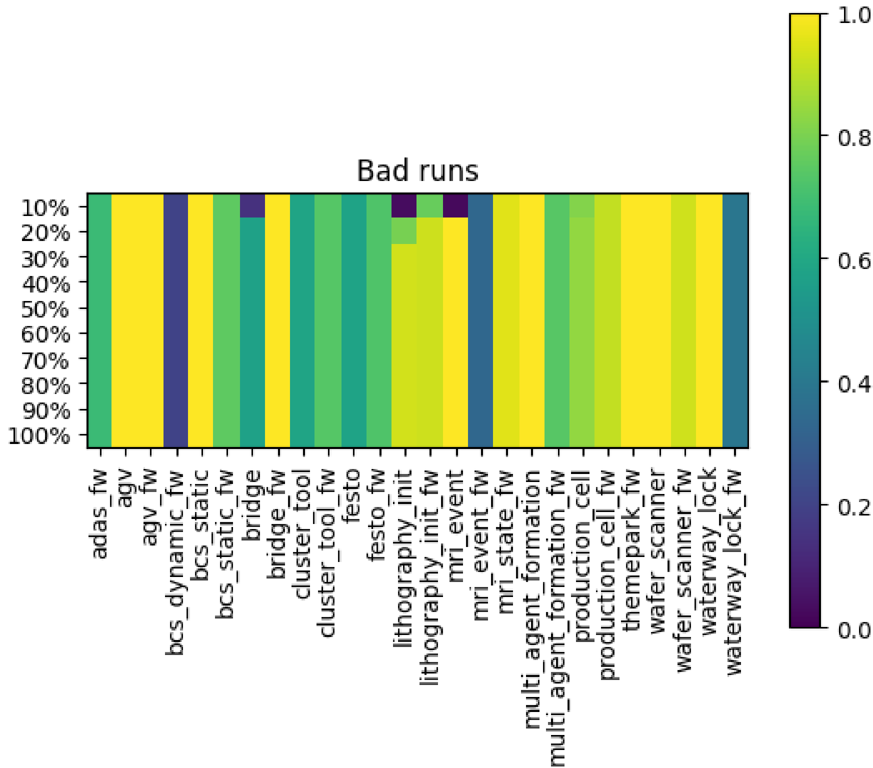


Figure 30: Plot of the performance of the *bad* runs of the *NCyclesPredictor* using a window of 3 cycles, with the threshold set to 0.1.

both of its constituent predictors to label the run as bad. The data reflects this: the predictor labels more runs as good, and fewer runs as bad. As a result, the predictor seems to score lower on average, but only by a single percent compared to the *NCyclesPredictor*. However, the predictor is able to label 3% more runs as good, so that it produces 3% fewer false positives. It is not quite as good on the bad runs, as it produces 4% more false negatives. Since we have decided to prioritize minimizing false positives, we conclude that this predictor outperforms the *NCyclesPredictor*.

Secondly, we attempt to combine the predictors using the *DisjunctionPredictor*. This predictor predicts that a run is bad when one of its constituent predictors classifies it as such. Again, we take the combination of the *TotalDuplicatesPredictor* and the *NCyclesPredictor*. After having seen 30% of the run, this combination scores 0.81 on average, 0.77 on the good runs and 0.86 on the bad runs. Therefore, it produces fewer false negatives than the *ConjunctionPredictor*, but more false positives. Since we decided that false positives are worse than false negatives, we still prefer the *ConjunctionPredictor* over this combination of predictors.

Lastly, we have a look at the *MultiPredictor*. Here, we take the combination of three predictors, and classify the result according to the majority of the predictors. Again, we include the *TotalDuplicatesPredictor* and the *NCyclesPredictor*. Since it does not make much sense to include a copy of one of the two predictors, but with different settings, we take the *VeryLittleProgressPredictor* as our third predictor, with a threshold of 1. The results show that this predictor performs similarly to the *ConjunctionPredictor*. This means that the *VeryLittleProgressPredictor* does not influence the predictions much. Since the *ConjunctionPredictor* is simpler, we prefer that predictor over the *MultiPredictor*. In the end, the *ConjunctionPredictor* consisting of the *NCyclesPredictor* and the *TotalDuplicatesPredictor* is the best predictor: it strikes the best balance between early detection of bad runs and having few false positives.

Now that we have identified the *ConjunctionPredictor* to be the best predictor, we can again look at its performance on the individual models. Figure 31 shows the average performance of the predictor on the benchmark models. Again, the *ConjunctionPredictor* performs the worst on the same models as the *NCyclesPredictor*: the `bcs dynamic fw`, `bridge fw` and `mri event fw` models.

When looking at the good runs only, as shown in Figure 32, we can see that the *ConjunctionPredictor* improves upon the *NCyclesPredictor* for the `lithography init`, `lithography init fw` and `mri event` models. Apparently the extra requirement by the *TotalDuplicatesPredictor* causes several more runs to be correctly classified as good, reducing the number of false positives.

When only the bad runs are considered, as shown in Figure 33, the *ConjunctionPredictor* performs worse than the *NCyclesPredictor*. This makes sense, as the *TotalDuplicatesPredictor* adds an additional requirement

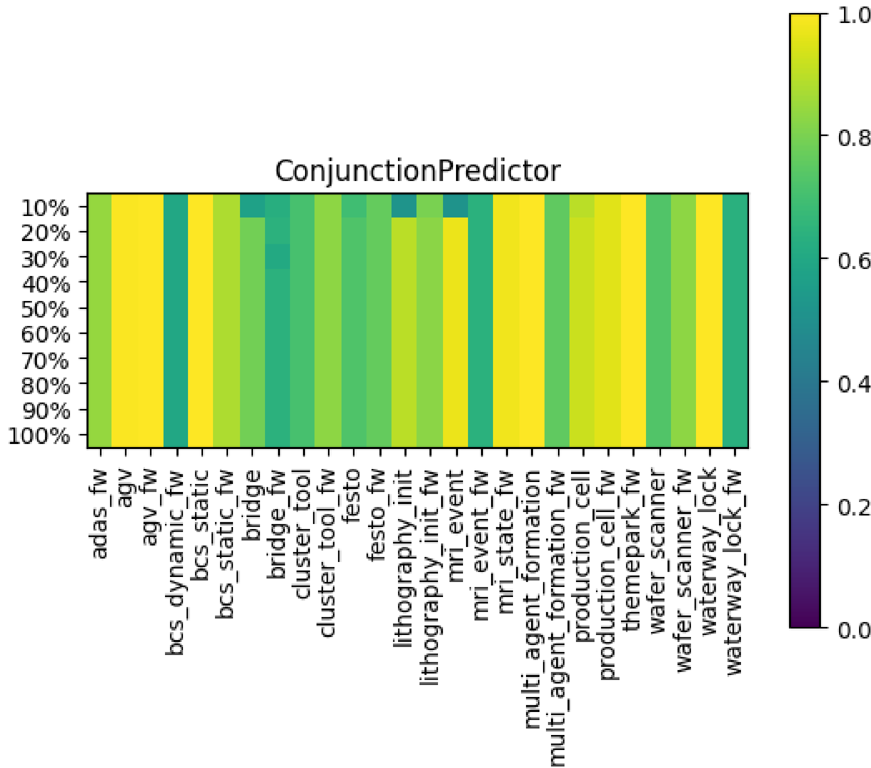


Figure 31: Performance of the *ConjunctionPredictor* on the individual benchmark models.

for deciding that a run is bad. The models where the *NCyclesPredictor* already performs well are also classified correctly by the *ConjunctionPredictor*, such as the `wafer scanner`, `multi agent formation` and `agv` models. However, bad runs for the models that the *NCyclesPredictor* struggles with are also difficult for the *ConjunctionPredictor*, such as the `festo` and `cluster tool` models. This is due to the fact that the *ConjuctionPredictor* is more strict than the *NCyclesPredictor*, and can in fact only label a run as *bad* when the *NCyclesPredictor* does so. However, as mentioned before, this is not a big problem, as the *ConjunctionPredictor* on average scores similar to the *NCyclesPredictor*, while producing fewer false positives.

5.5 Predictor validation

Now that we have found and tuned the *ConjunctionPredictor*, we need to validate that it indeed performs well. The predictor has good performance on the dataset used for tuning, but it is not fair to draw conclusions from this. In fact, we have chosen this predictor such that is has good performance on the tuning dataset. In order to be able to say anything about its effectiveness, we need to validate the predictor on a separate dataset. For this dataset, called the validation set, we run experiments similar to the ones used for the tuning set, except on entirely different models. This way, there is no crossover between the tuning set and the validation set, so that the results are not influenced by the data in the tuning set. For the validation set, we choose several larger models to collect the data from. This way, we have a representative scenario for validating the predictor, as in practice the user will be mostly interested in using the predictor when they are actually synthesizing models that take longer than a few seconds to synthesize. Table 8 shows details on the models in this validation set. There is no reference for the `fifo mixed` model, but it is a variant on a CIF example[7]. Note that the validation set contains three models that were generated by the assembler described in [40]. It is future work to experiment with a more diverse set of models. In total, the validation set contains 238 good runs and 168 bad runs.

The results of the predictor on this validation set can be seen in Figure 34. On average, having seen 20% of a run, it can classify 82% of the runs correctly. It classifies 68% of good runs correctly at this point, and 96% of bad runs. It performs well on the `Wwl1`, `Wwl2` and `Wwl3` models. It performs the worst on the `Wafern2_fw`, `WWl2_fw` and `Wwl3_fw` models. When we again separate the data into good and bad runs, we get a more complete overview of the problem. As seen in Figure 35, the predictor seems to have difficulties predicting good runs for all three of the aforementioned models. When viewing the data more closely, we find that these models only have runs in the dataset with many duplicate node creations. This appears to be a characteristic of several models that have forward reachability enabled. The extra reachability computation seems to produce many

[7]See also `https://eclipse.dev/escet/cif/synthesis-based-engineering/example.html`.

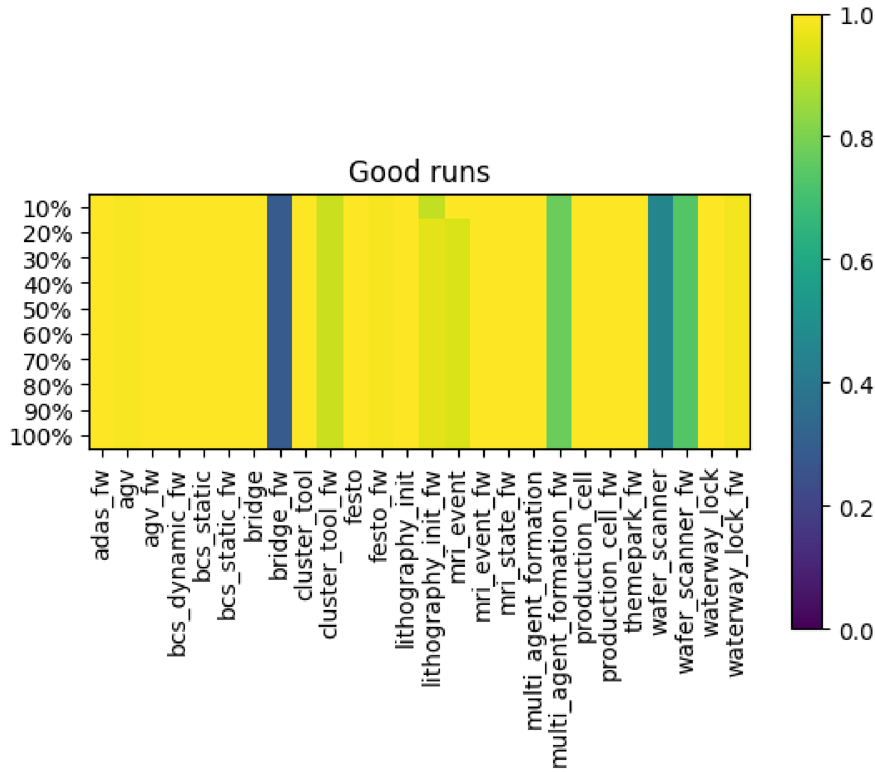


Figure 32: Performance of the *ConjunctionPredictor* on the good runs of the individual benchmark models.

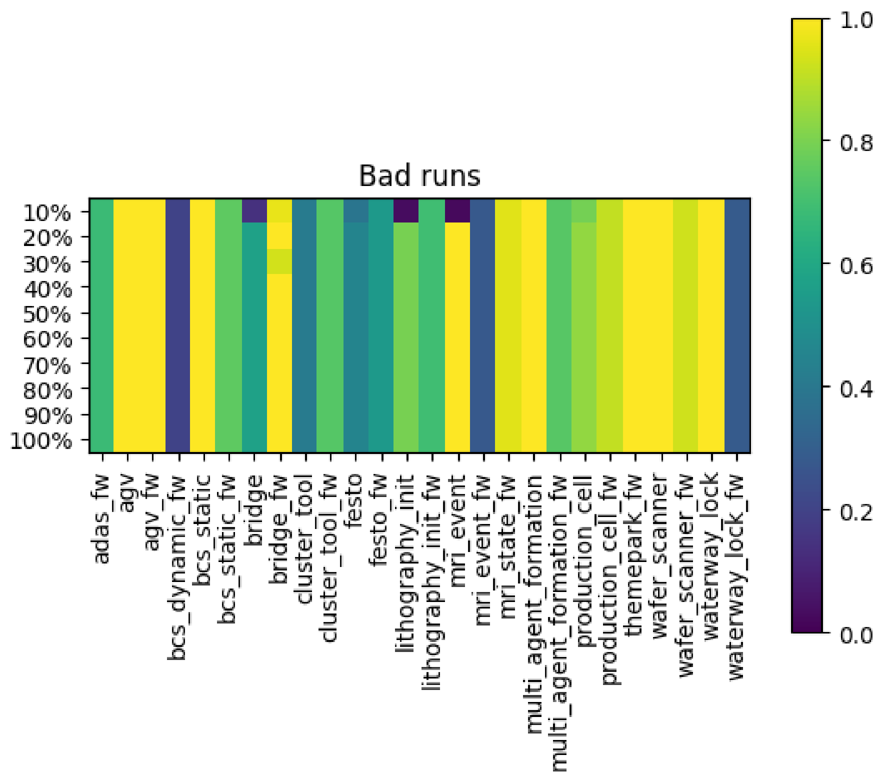


Figure 33: Performance of the *ConjunctionPredictor* on the bad runs of the individual benchmark models.

Table 8: Overview of the models used for the validation set.

Nr	Short name	Full name	Refs	Size (US)	Size (CS)
1	fifo-mixed	FIFO robot with mixed conditions	-	$4.54 \cdot 10^6$	$1.33 \cdot 10^2$
2	wafer-scanner-n2	Wafer scanner (n=2)	[38]	$9.17 \cdot 10^7$	$1.59 \cdot 10^7$
3	wwl1	Waterway lock 1	[40]	$5.78 \cdot 10^{69}$	$2.71 \cdot 10^{63}$
4	wwl2	Waterway lock 2	[40]	$8.69 \cdot 10^{81}$	$5.34 \cdot 10^{76}$
5	wwl3	Waterway lock 3	[40]	$5.77 \cdot 10^{83}$	$1.53 \cdot 10^{78}$

duplicates in some cases, which causes the labeling of the data to be skewed. It is possible that there are no better runs for these models, or that we simply have not found an actual good run for these models. When looking at the bad runs, as shown in Figure 36, the performance of the predictor is very good, as it is able to detect bad runs after having only seen 20% of the run. All in all, the predictor performs well, as it is able to predict 82% of runs correctly after having seen 20% of those runs, although there are some notable exceptions that should be investigated further.

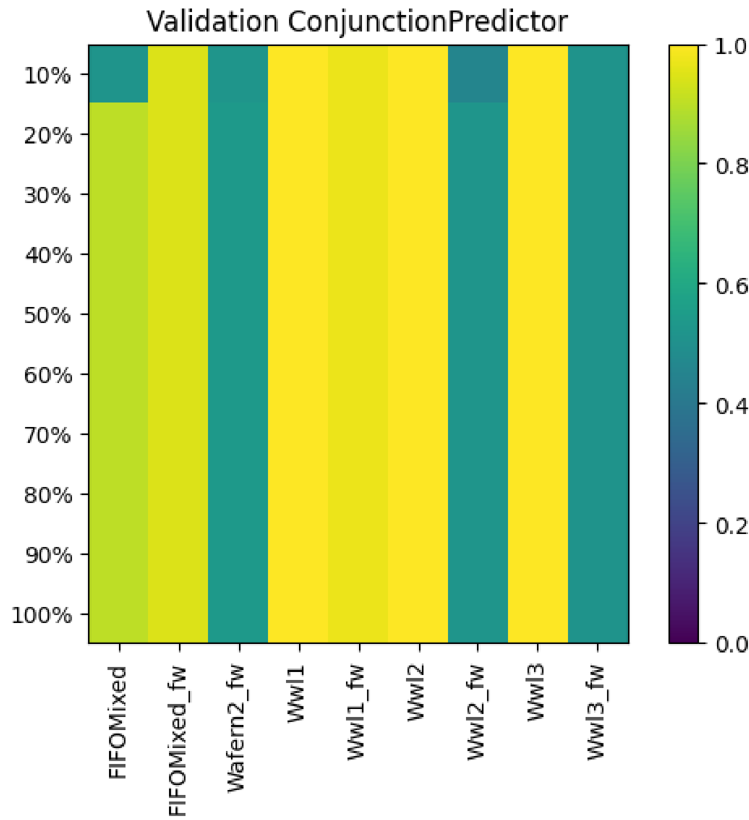


Figure 34: Performance of the *ConjunctionPredictor* on the validation dataset. Note that `Wafern2` is not in the set, as the model did not synthesize within reasonable time for any of the chosen variable orders.

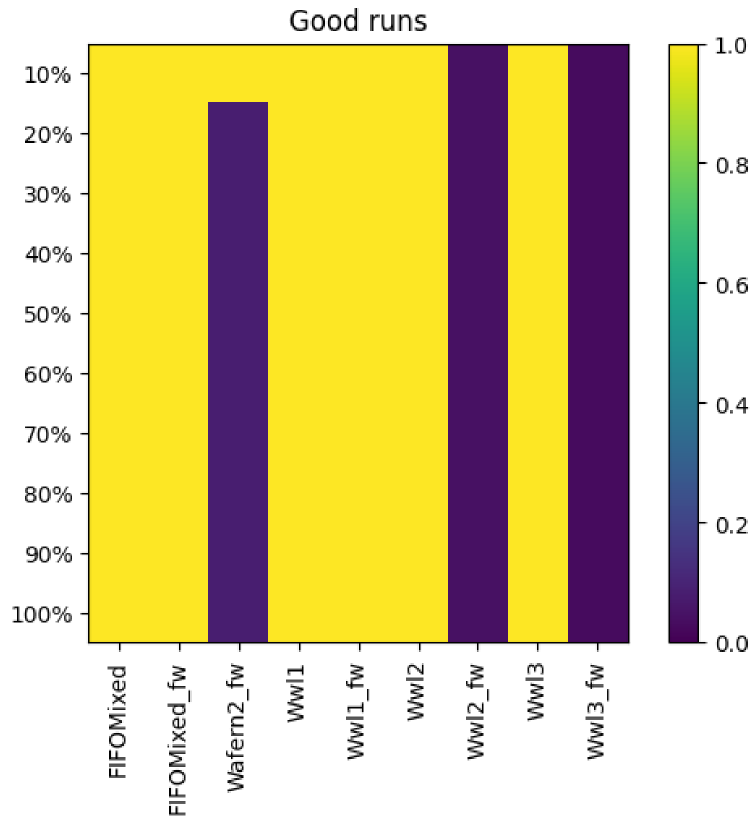


Figure 35: Performance of the *ConjunctionPredictor* when classifying good runs from the validation dataset.

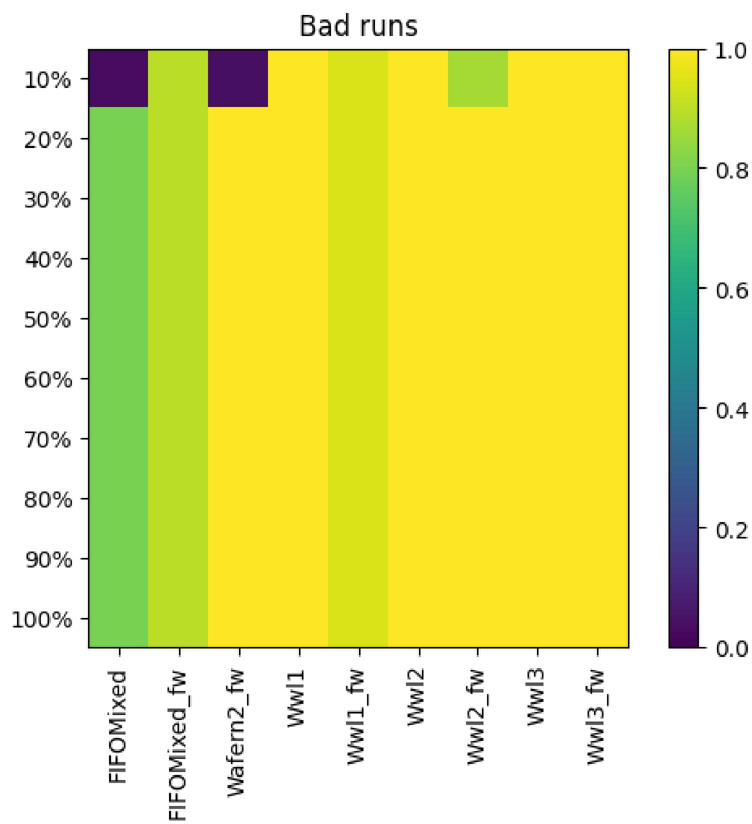


Figure 36: Performance of the *ConjunctionPredictor* when classifying bad runs from the validation dataset.

6 Conclusion

When performing supervisory controller synthesis, the user has no indication whether they should wait until the synthesis algorithm terminates, or try running the algorithm using different settings instead. In this thesis we explored whether it is possible to let the user know when a particular set of settings for the synthesis algorithm performs poorly. In order to investigate this, we looked at the symbolic saturation algorithm used in the CIF data-based synthesis tool from three perspectives: the input of the algorithm, the inner workings of the algorithm and the data structures used to implement the algorithm. We used a step-wise approach to investigate each of these three perspectives in parallel, in order to try and find some indicator of when a set of settings is bad. When a perspective did not provide enough inside, we stopped further investigation of that perspective and focused on the others. Out of the three perspectives, the data structures perspective turned out to be the most promising, as it revealed that the current implementation of the saturation algorithm in the JavaBDD library leads to a lot of duplicate work being done. In fact, many BDD nodes were created multiple times. These duplicate nodes turned out to be the indicator that we were looking for.

Next, we came up with a set of predictors that predict whether a given run will be bad. A good predictor should inform the user of a bad run early, but should not produce many false positives. We tried the *TotalDuplicatesPredictor*, *NCyclesPredictor*, *NoProgressPredictor* and *VeryLittleProgressPredictor*, which we could also combine using the *ConjunctionPredictor*, *DisjunctionPredictor* and *MultiPredictor*. Using a dataset consisting of data on duplicate node creations from synthesis runs on different models with different settings, we found that the *ConjunctionPredictor* performed best out of all tested predictors, with a good balance of detecting many bad runs and having very few false negatives. This *ConjunctionPredictor* itself consists of the *NCyclesPredictor* with a window size of 3 and a threshold of 0.1, and the *TotalDuplicatesPredictor* with a threshold of 0.1. Finally, this predictor was validated on a separate validation dataset, where it correctly classified a run as good or bad with 82% accuracy, after having seen only 20% of a run. It does produce false positives, as it correctly classifies good runs with 68% accuracy, after having seen 20% of the run, although these false positives are only produced on specific models.

7 Future work

Since the investigation performed in this thesis is exploratory, there are many possible avenues for future work. We describe several of these possibilities here. Some of these are improvements over the current work, while others are extensions to this work.

Firstly, we can continue with the two perspectives that did not lead to an indicator of progress. For the transition relation perspective, we could further consider the possibility of a velocity for the saturation algorithm. In this thesis, we attempted to find this velocity using a linear fit. However, perhaps a different fit that takes the oscillations of the algorithm into account would be more appropriate.

For the input perspective, the blow-up mentioned in Section 4.3 still feels like a possible indicator for slowdown of progress, since the BDD increases drastically in size. To study this further, we could look at the number of nodes of the BDD over the course of a synthesis run. If this increases sharply, then every relational product will take longer to compute. It still does not say much about how long it takes until a node is saturated, however, so that more work on this is indeed needed.

Furthermore, we can combine the ideas of the two perspectives, by keeping track of where the transition relations are applied in the saturation algorithm. However, this also requires us to keep track of the intermediate BDDs, as the transition relations can be applied to nodes that are not in the initial BDD. Perhaps this would provide more insight into the workings of the algorithm, compared to separate experiments for both perspectives.

The other perspective that we studied in this thesis was the implementation of the saturation algorithm in JavaBDD. Here, we found that the implementation causes many duplicate nodes and duplicate cache entries to be created. There are many avenues for future work when it comes to these duplicate computations.

When studying some of the figures on duplicate node measurements, we noticed that some duplicate nodes could be measured in the very first garbage collection cycle. This does not make much sense, however. This can be attributed to hash collisions when calculating the identifiers of the nodes. We should investigate how many hash collisions are produced exactly, and whether there is a better hashing scheme available to avoid these collisions.

Another improvement can be made when it comes to measuring the memory usage of the duplicate node measurements. As mentioned in Section 5.1, the impact of the duplicate node measurements on the memory usage of a run of the synthesis algorithm is left as future work. Memory measurements in Java are tricky, as the Java Virtual Machine manages its own memory. Since it can also free memory during a run, it is possible that more memory is in use before a run than after all the measurements are performed. Therefore, an investigation into how to properly measure the memory usage that the measurements take is needed.

Another possibility is to use the insights obtained in this perspective to improve the implementation of the reachability algorithm itself. It would be interesting to see whether it is possible to reduce the number of duplicate computations. One idea that may be interesting here would be to change the implementation of the saturation algorithm to saturate nodes iteratively instead of recursively, using a queue to store computations that still have to be performed (note that the current implementation is recursive and therefore uses the call-stack). This may influence the number of duplicate computations performed as the order of the computations changes. Alternatively, some other order of saturating BDD nodes could help reduce the amount of duplicate work. Another improvement could be to split the operation cache table into multiple tables. JavaBDD already does this for several operations, but all saturation and relational product operations are stored in the same cache table. Instead, multiple separate tables can be used, which would reduce the number of cache overwrites. Alternatively, some sort of heuristic could be made to indicate which cache entries are most important, such that important entries can be protected against being overwritten.

After we had studied the three perspectives, we decided to continue with the duplicate work measurements to create a predictor for when a run is bad. To do this, we first generated data on good and bad runs, by changing the settings of the synthesis algorithm. Here, we have mostly used the variable ordering settings to create good and bad runs. However, the node and cache table size settings could also be interesting. One could track the differences in cache entries created when altering the size of the table. This experiment could be repeated with different sizes for the cache table, after which the operations performed in each experiment could be compared. This could provide an insight in how these settings influence the saturation algorithm.

After the data was collected, we modeled the runtime of each run using a linear combination of the number of operations performed and the number of garbage collection cycles performed, which we used to label the runs. However, it is not clear that this linear combination is the best possible way to model the runtime of the algorithm. Perhaps there is a more complex model that works better.

The process of labeling should also be looked into further. The labeling as done in this work is based on the best run of a given model. However, it is not obvious that the best run of that model is actually in the produced dataset. In fact, there have been models where every run produces many duplicate nodes. It is unclear whether there actually exist runs that perform better. Therefore, some independent labeling strategy could be worth investigating further.

After labeling, we used the data to come up with several progress predictors. The *ConjunctionPredictor*

that we have come up with can be improved further. As it stands, it still produces many false positives and false negatives. Ideally, it should have an accuracy that is way higher earlier on in a run of the synthesis algorithm. In order to be able to determine this properly, we should reconsider how we divide a run into smaller parts. As it stands, we have used increments of 10%, and evaluated the predictor at every such increment. However, it is more interesting to the user to see after how much time a stable prediction can be made. This could then be done both in terms of absolute time, as well as in the number of garbage collection cycles performed. Lastly, it would be interesting to know how long it takes to perform a good run in comparison to how long it takes until a stable prediction can be made.

We should also still consider the possibility that there could be an alternative predictor that is more accurate than the *ConjunctionPredictor*. However, to make stronger claims about the accuracy of such a predictor, the predictor should also be tested on even more models. Lastly, we should investigate why predictions on some models are more difficult than others.

The validation set used in validating the settings for the *ConjunctionPredictor* contains three models of waterway locks, generated by an assembler. However, this could incur bias. We should create a larger validation set using more diverse models.

Finally, we could look into how to make the predictors more practical. Firstly, the predictors should be integrated into the Eclipse ESCET toolkit, to make sure users can actually use them. In fact, since the prediction method requires only that BDDs and the saturation algorithm are used, it may also be integrated into other tools that use these techniques. The integration of the predictors as-is may not be too useful, however, as the predictors studied in this thesis are still very primitive. After all, when a run is bad, they can indicate that it is bad, but not what settings should be changed to improve it. It would be interesting to come up with a method of deciding which settings should be changed. This information would be very helpful to the user, as they now still have to do trial-and-error in order to improve the run. In order to realize this, we need more knowledge on the influence of the available settings on the performance of the algorithm.

We should also investigate how the progress can be communicated to the user. It could be as simple as a message that says the user should terminate the run, but that is not very informative. Another possibility would be to show a graph similar to the duplicate node plots, so that the user can see what the prediction is based on. We should study which manner of communicating is most informative to the user.

Overall, in this thesis only a first step is taken towards reporting ‘synthesis progress’, and more work is needed before this can be made practical.

References

[1] W. J. Fokkink, M. A. Goorden, D. Hendriks, D. A. van Beek, A. T. Hofkamp, F. F. H. Reijnen, L. F. P. Etman, L. Moormann, J. M. van de Mortel-Fronczak, M. A. Reniers, J. E. Rooda, L. J. van der Sanden, R. R. H. Schiffelers, S. B. Thuijsman, J. J. Verbakel, and J. A. Vogel. Eclipse ESCET™: The Eclipse Supervisory Control Engineering Toolkit. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 44–52, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-30820-8.

[2] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992. ISSN 0360-0300. doi: 10.1145/136035.136043.

[3] Dennis Hendriks, Michel Reniers, Wan Fokkink, and Wytse Oortwijn. Overview and Performance Evaluation of Supervisory Controller Synthesis with Eclipse ESCET v4.0, 2025. URL `https://arxiv.org/abs/2511.04370`.

[4] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli. Dynamic variable reordering for BDD minimization. In *Proceedings of EURO-DAC 93 and EURO-VHDL 93- European Design Automation Conference*, pages 130–135, 1993. doi: 10.1109/EURDAC.1993.410627.

[5] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An Efficient Iteration Strategy for Symbolic State—Space Generation. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 328–342, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45319-2.

[6] Tom van Dijk and Jaco van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19(6):675–696, Nov 2017. ISSN 1433-2787. doi: 10.1007/s10009-016-0433-2.

[7] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *30th ACM/IEEE Design Automation Conference*, pages 272–277, 1993. doi: 10.1145/157485.164890.

[8] Tom van Dijk, Robert Wille, and Robert Meolic. Tagged BDDs: Combining reduction rules from different decision diagram types. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 108–115, 2017. doi: 10.23919/FMCAD.2017.8102248.

[9] Gianfranco Ciardo, Andrew Miner, Lichuan Deng, and Junaid Babar. RexBDDs: Reduction-on-Edge Complement-and-Swap Binary Decision Diagrams. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706011. doi: 10.1145/3649329.3656533.

[10] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996. doi: 10.1109/12.537122.

[11] Radu I. Siminiceanu and Gianfranco Ciardo. New Metrics for Static Variable Ordering in Decision Diagrams. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 90–104, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33057-8.

[12] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, GLSVLSI '03, page 116–119, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136773. doi: 10.1145/764808.764839.

[13] Sam Lousberg, Sander Thuijsman, and Michel Reniers. DSM-based variable ordering heuristic for reduced computational effort of symbolic supervisor synthesis. *IFAC-PapersOnLine*, 53(4):429–436, 2020. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.2021.04.058. 15th IFAC Workshop on Discrete Event Systems WODES 2020 — Rio de Janeiro, Brazil, 11-13 November 2020.

[14] Lucien Ouedraogo, Ratnesh Kumar, Robi Malik, and Knut Akesson. Nonblocking and Safe Control of Discrete-Event Systems Modeled as Extended Finite Automata. *IEEE Transactions on Automation Science and Engineering*, 8(3):560–569, 2011. doi: 10.1109/TASE.2011.2124457.

[15] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. *Ten Years of Saturation: A Petri Net Perspective*, pages 51–95. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-29072-5. doi: 10.1007/978-3-642-29072-5_3.

[16] Fabio Somenzi. Efficient manipulation of decision diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):171–181, May 2001. ISSN 1433-2779. doi: 10.1007/s100090100042.

[17] Parosh Abdulla, Per Bjesse, and Niklas Een. Symbolic Reachability Analysis Based on SAT-Solvers. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1785, pages 411–425, 03 2000. ISBN 978-3-540-67282-1. doi: 10.1007/3-540-46419-0_28.

[18] Shai Haim. *Runtime Estimation of Backtracking Satisfiability Solvers: A Machine Learning Approach.* PhD thesis, The University of New South Wales, 2010.

[19] K. Ravi and F. Somenzi. Efficient fixpoint computation for invariant checking. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*, pages 467–474, 1999. doi: 10.1109/ICCD.1999.808582.

[20] Arash Vahidi, Martin Fabian, and Bengt Lennartson. Efficient supervisory synthesis of large systems. *Control Engineering Practice*, 14(10):1157–1167, 2006. ISSN 0967-0661. doi: https://doi.org/10.1016/j.conengprac.2006.02.013. The Seventh Workshop On Discrete Event Systems (WODES2004).

[21] Mohamed Raseen, P.W. Chandana Prasad, and Ali Assi. An efficient estimation of the ROBDD's complexity. *Integration*, 39(3):211–228, 2006. ISSN 0167-9260. doi: https://doi.org/10.1016/j.vlsi.2005.06.002.

[22] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2013.10.003.

[23] D. Bremer. Artifact for the master's thesis 'Should I wait for supervisory controller synthesis to finish or change the settings?', 2026.

[24] Sander Thuijsman, Dennis Hendriks, and Michel Reniers. Reducing the computational effort of symbolic supervisor synthesis. *Discrete Event Dynamic Systems*, 34(4):689–732, Dec 2024. ISSN 1573-7594. doi: 10.1007/s10626-024-00403-4.

[25] Tim Korssen, Victor Dolk, Joanna M. van de Mortel-Fronczak, Michel A. Reniers, and Maurice Heemels. Systematic Model-Based Design and Implementation of Supervisors for Advanced Driver Assistance Systems. *IEEE Transactions on Intelligent Transportation Systems*, 19(2):533–544, 2018. doi: 10.1109/TITS.2017.2776354.

[26] Maurice H. ter Beek, Michel A. Reniers, and Erik P. de Vink. Supervisory Controller Synthesis for Product Lines Using CIF 3. *Proc. 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA), Part I*, 9952:856–873, 2016. doi: 10.1007/978-3-319-47166-2_59.

[27] Sander B. Thuijsman and Michel A. Reniers. Supervisory control for dynamic feature configuration in product lines. *ACM Transactions on Embedded Computing Systems*, 2023. doi: 10.1145/3579644.

[28] F. F. H. Reijnen, M. A. Reniers, J. M. van de Mortel-Fronczak, and J. E. Rooda. Structured synthesis of fault-tolerant supervisory controllers. *Proc. 10th Symposium on Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS), IFAC-PapersOnLine*, 51(24):894–901, 2018. doi: 10.1016/j.ifacol.2018.09.681.

[29] Rong Su, Jan H. van Schuppen, and Jacobus E. Rooda. Aggregative synthesis of distributed supervisors based on automaton abstraction. *IEEE Transactions on Automatic Control*, 55(7):1627–1640, 2010. doi: 10.1109/TAC.2010.2042342.

[30] F. F. H. Reijnen, M. A. Goorden, J. M. van de Mortel-Fronczak, M. A. Reniers, and J. E. Rooda. Application of dependency structure matrices and multilevel synthesis to a production line. *Proc. 2nd Conference on Control Technology and Applications (CCTA)*, pages 458–464, 2018. doi: 10.1109/CCTA.2018.8511449.

[31] Zowi Vos. Initialization and Termination of Flexible Manufacturing Systems: A formal approach. Master's thesis, Eindhoven University of Technology, 2020.

[32] Sander B. Thuijsman, Michel A. Reniers, and Dennis Hendriks. Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis. *Proc. 17th Conference on Automation Science and Engineering (CASE)*, pages 777–783, 2021. doi: 10.1109/CASE49439.2021.9551593.

[33] R. J. M. Theunissen, M. Petreczky, R. R. H. Schiffelers, D. A. van Beek, and J. E. Rooda. Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner. *IEEE Transactions on Automation Science and Engineering (TASE)*, 11(1):20–32, 2014. doi: 10.1109/TASE.2013.2279692.

[34] Rolf J. M. Theunissen. *Supervisory Control in Health Care Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2015. URL `https://research.tue.nl/en/publications/supervisory-control-in-health-care-systems/`.

[35] Kai Cai and W. M. Wonham. New results on supervisor localization, with case studies. *Discrete Event Dynamic Systems*, 25:203–226, 2015. doi: 10.1007/s10626-014-0194-6.

[36] Lei Feng, Kai Cai, and W. M. Wonham. A structural approach to the non-blocking supervisory control of discrete-event systems. *The International Journal of Advanced Manufacturing Technology*, 41:1152–1168, 2009. doi: 10.1007/s00170-008-1555-9.

[37] Stefan T. J. Forschelen, Joanna M. van de Mortel-Fronczak, Rong Su, and Jacobus E. Rooda. Application of supervisory control theory to theme park vehicles. *Discrete Event Dynamic Systems*, 22:511–540, 2012. doi: 10.1007/s10626-012-0130-6.

[38] L. J. van der Sanden, Michel A. Reniers, Marc C. W. Geilen, A. A. Basten, Johan Jacobs, Jeroen P. M. Voeten, and Ramon R. H. Schiffelers. Modular model-based supervisory controller design for wafer logistics in lithography machines. *Proc. 18th Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 416–425, 2015. doi: 10.1109/MODELS.2015.7338273.

[39] F. F. H. Reijnen, M. A. Goorden, J. M. van de Mortel-Fronczak, and J. E. Rooda. Supervisory control synthesis for a waterway lock. *Proc. 1st Conference on Control Technology and Applications (CCTA)*, pages 1562–1563, 2017. doi: 10.1109/CCTA.2017.8062679.

[40] Marzhan M. Baubekova, Koen J. van Eldik, Joanna M. van de Mortel-Fronczak, Wan J. Fokkink, and Jacobus E. Rooda. SBE configurator: A model generation tool for synthesis of ship lock supervisors. *IFAC-PapersOnLine*, 58(1):288–293, 2024. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.2024.07.049. 17th IFAC Workshop on discrete Event Systems WODES 2024.