Universiteit Leiden

Certifying constraints from hardware designs

Enzo Beekman (s3331644)

Supervisors:
Emily Yu & Nils Froleyks

Bachelor Thesis– Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

July 9, 2026

Abstract

In industrial hardware designs it is common to have multiple properties per model. Each of these properties needs to be model checked and certified to prove the property holds. In this thesis, adding proven properties as constraints on the model is explored as an option to improve model checking and certification efficiency. This thesis aims to implement a tool to automate this process. Using different software and constructions, the time to model check and certify a witness decreases. This tool also measures the time it takes to model check and certify different models with different amount of constraints. This is used to experimentally measure the difference in time between different approaches. This thesis experimentally demonstrates the effectiveness of adding proved properties as model constraints by using benchmarks from a hardware model competition.

Contents

1 Introduction

Hardware designs are everywhere. From processors to embedded systems, they form an important pillar for modern technology, such as civilian aircraft systems [8]. That is why verification of these designs is necessary to make sure they work as expected before they are manufactured. The assessment of a hardware circuit (model) such as in figure 1 is in simple terms done by exploring every possible state the model can find itself in. Models themselves can be represented with a

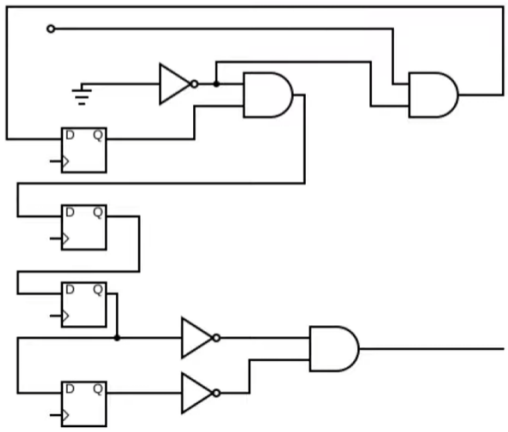

Figure 1: A simple circuit.

symbolic representation called AIG [5]. This stands for And-inverter Graph. Using this format, a model checker determines if a model is safe. The model checker will provide a proof of what it has found. This is a prove that the model is safe or unsafe. Giving a proof is a way for other software to check if the model checker was correct and did not make a mistake, which may happen with software that is complex. These simple steps make up the basis of hardware model checking.

1.1 The problem

To see if a circuit is safe or to see if it behaves like expected, a circuit can have a bad state signal. This is an output wire in the circuit that may never be true (in binary 1) [6]. A model checker can use this bad state signal to test if the hardware design is working correctly. In large hardware designs it is common to have multiple of these bad state wires. Every bad state would then be representing a small portion of the circuit. This still means a model checker has to take into account every state a circuit can be in. To prevent this, a constraint can be introduced. A constraint is an assumption about a circuit saying that a certain wire will never be true or false. This constraint can be used to speed up model checking. Once a model checker is done and found that the bad state wire is not being violated, it will have given a proof that the property will always hold for this circuit. This can then in turn be used as a constraint for that same circuit. This paper aims to optimize model checking by using the steps and tools discussed in this chapter.

1.2 Thesis overview

This thesis aims to answer the following research question : Can constraint extraction in hardware model design be used to efficiently certify multiple safety properties as model constraints?
Section 2 talks about the work that lead to the research question of this thesis. In section 3 an overview of the theory and tools being used is given. In section 4 will be a description of the technical approach behind the tool being used for running experiments. Section 5 will talk about the exact setup and results of the experiments. The thesis finishes with section 6 where the most significant results are discussed and a short talk about further research.

2 Related Work

The first part of this thesis will discuss formal definitions for a circuit and a witness circuit. In addition to that, the definition for a bad state property and the underlying theorem behind AIGMERGE (see section 3) will be discussed.
The first formal definition is for a circuit, given by Definition 1 [7].

Definition 1 (Circuit) *A circuit M = (I,L,R,F,P,C) is defined as a tuple consisting of the following attributes:*

1. *I: a finite ordered set of Boolean input variables;*
2. *L: a finite ordered set of Boolean latch variables;*
3. *$R = \{r_l(I, L)|l \in L\}$, where $r_l(I, L)$ is a reset function for latch l;*
4. *$F = \{f_l(I, L)|l \in L\}$, where $f_l(I, L)$ is a transition function for latch l;*
5. *P(I,L) represents the set of good states; and*
6. *C(I,L) encodes the set of constrained states.*

According to Definition 1, a circuit contains inputs, latches and states. Attribute 1 and 2 make it clear that the set of inputs and latches is an ordered set, meaning they have a set order. The reset function in attribute 3 means a function resetting a latch to its original state and attribute 4 defines that a latch has a function defining its behavior for each clock cycle. A state of the circuit is an assignment of inputs and latches satisfying the constraint.
For this thesis the focus is on bad state properties, which are also known as safety properties. The set of initial states is defined by $R\{L\} = \wedge_{l\in L}(l \leftrightarrow r_l(I, L)$. The same notation is used for transitions $F_{0,1}\{L\} = \wedge_{l\in L}(l_1 \leftrightarrow f_l(I_0, L_0)$ A circuit is safe if the property P holds in all states. If this is not the case, it means there is a counterexample. This counterexample is a path from a reset state to a bad state where all states in between satisfy the constraint. This is formalized as:

$$R_0\{L\} \wedge \bigwedge_{i\in[0,n)} F_{i,i+1}\{L\} \wedge \bigwedge_{i\in[0,n]} C_i \wedge \neg P_n$$

If a path exists that satisfies these conditions and still leads to a bad state, it is called unsafe. If a circuit is safe, a certificate needs to be made to prove the safety of the circuit. This comes in the form of a witness circuit. In Definition 2, each condition is encoded as a SAT-formula. According to the definition, R' needs to be stratified. A circuit is said to be stratified if the syntactic dependency graph induced by its reset function is acyclic. This means that there are no cyclic dependencies among the reset functions of the witness circuit.

Definition 2 (Witness Circuit) *$W = (I', L', R', F', P', C')$ is a witness circuit of circuit $M = (I, L, R, F, P, C)$, both with constraints, if R' is stratified and for $K = L \cap L'$:*

1. *Reset:* $R\{K\} \wedge C \rightarrow R\{K\} \wedge C'$;
2. *Transition:* $F_{0,1}\{K\} \wedge C_0 \wedge C_1 \wedge C'_0 \rightarrow F'_{0,1}\{K\} \wedge C'_1$;
3. *Property:* $(C \wedge C') \rightarrow (P' \rightarrow P)$;
4. *Base:* $R'\{L'\} \wedge C' \rightarrow P'$;
5. *Step:* $P'_0 \wedge F'_{0,1}\{L'\} \wedge C'_0 \wedge C'_1 \rightarrow P'_1$;

Intuitively, a witness circuit is a proof of correctness. The model checker provides a circuit that can be verified more easily then the original model. This way, other software does not have to do complicated calculations like the model checker does, but can verify the work using the 5 checks described in Definition 2.

The conditions in Definition 2 are being used by a certifying program to test the validity of the certificate. The witness circuit simulates part of the model and has an inductive invariant. An inductive invariant is a property that proves a property holds.
Constraint extraction can simplify verification for model checking. The problem is that the witness circuit produced applies to the preprocessed circuit and not to the base circuit. To discuss this challenge, a model constraint is first defined in Definition 3.

Definition 3 (Model Constraint) *Given a circuit $M = (I, L, R, F, P, C)$, the formula $D(I, L)$ is said to be a model constraint if it holds in all reachable states, i.e., the following formula is unsatisfiable for any n:*

$$R_0\{L\} \wedge \bigwedge_{i \in [0,n)} F_{i,i+1}\{L\} \wedge \bigwedge_{i \in [0,n]} C_i \wedge \neg D_n$$

A model constraint is a constraint that holds in all reachable state, like a safety property. If a model constraint is proven, it can simply be combined with existing constraints. This makes model constraints useful, as they do not alter the set of reachable states, but can be used to constraint the model when looking at other bad state properties.
When adding a model constraint to a circuit, a model checking engine will produce a witness for that new circuit. To do this, it needs to be composed with a witness for the model constraint D. In Theorem 1, it is assumed that inputs and latches not shared with the model are unique to the respective witness circuits. if this is not the case, renaming is necessary. Theorem 1 describes how to post-process a witness circuit that has an added model constraint D.

Theorem 1 *Let $M = (I, L, R, F, P, C)$ be a model with constraint D, and let $M_P = (I, L, R, F, P, C \wedge D)$ be the constrained model with witness $W_P = (I^P, L^P, R^P, F^P, P^P, C^P)$. Let $W_D = (I^D, L^D, R^D, F^D, D^D, C^D)$ be a witness certifying that D is a model constraint, i.e., a witness for $M_D = (I, L, R, F, D, C)$. If the inputs and latches of W_P and W_D are disjoint except for those shared with M, then the combined witness circuit W defined as follows is a witness for M.*

$$W = (I^P \cup I^D, L^P \cup L^D, R^P \cup R^D, F^P \cup F^D, P^P \wedge P^D, C^P \wedge C^D \wedge D)$$

In Theorem 1, it is described how a witness circuit is composed for a circuit that does not have a model constraint added. The produced witness W is then a valid certificate for the unconstrained model, which is therefore required without model checking that specific model. This construction can be used to certify multiple properties of the same model. This way multiple properties can be certified by composing multiple witness circuits for each property.

3 Background

3.1 Circuit components

A hardware design consists of a circuit/model. A circuit itself consists of different components. Each component is connected with wires. Wires have boolean values, which means that they are either 0 or 1. The first type of component is an input wire. An input wire can connect to other components and can be one of the two boolean values. This is useful to get different configurations of a circuit. Another component is the AND-gate. The AND-gate takes two wires as inputs and has one wire as output. The value of the output wire depends on the values of the input wires and is shown in figure 2. The last component is the latch. A latch has a single wire for both input and

AND gate truth table		
Input		Output
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2: truth table of an AND-gate component.

output. It holds a single boolean value. On the next clock cycle, it gives this value to its output wire and holds the value that is in the latch's input wire. A latch also has a reset value, which is the value a latch starts out with. Latches are useful, because they can represent the current state of the circuit. A circuit can also be represented graphically like in figure 1. The input wire is simple a line with a dot. The latch and AND-gate are represented with symbols that are shown in figure 3.

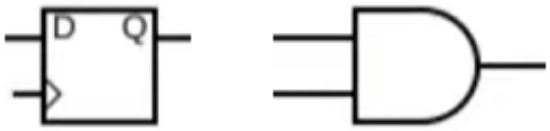

Figure 3: The graphical representation of a latch (left) and an AND-gate (right)

A circuit can also have a ground wire, which is always 0. To show the value of a wire being inverted, a triangle with a dot is used. The components described above can connect with each other by connecting the wires. This means that the output wire of one component is the same as an input wire from another.

3.2 And-inverter graph

To perform model checking on a circuit, a textual representation is needed. This representation comes in the form of an And-Inverter graph (AIG). An AIG is a text file containing numbers which represent different components like inputs, latches and and gates. The file starts with the header, which is the first line in the file. This header gives information about the amount of components , bad state properties and the constraints of the model. Figure 4 shows an example of an header file for a model that can be model checked. The first three word in the header tells a program what

```
aig 21222 468 1878 0 18876 1 2
```

Figure 4: An example of an header for an AIG file

kind of AIG file it is working with. For this thesis, only the words aig and aag are relevant. Their difference will be discussed below. The first number (from left to right) represents the maximum amount of variables that the circuit has. Variables are an individual input, latch or AND-gate. Every input, latch and AND-gate has a signal number and wire number. The signal number 0 is reserved for the ground state. The signal number represents a variable and a wire number represents the state of that variable and is equal to the signal number times two. For example, the second input has signal number 2 and wire number 4. If we want to give the inverted value of input 2 to an AND-gate, then we add one to the wire number. Therefore, wire numbers 4 and 5 both represent signal number 2. This eliminates the need for an explicit inverter node. The other numbers in the header represent (from left to right), the amount of inputs, latches, outputs, AND-gates, bad state property and constraints the circuit has. Other numbers after that are irrelevant for this paper. After this meta data in the header, there are the different sections detailing the structure of the circuit. The following explanation applies to the aag format, which is an ASCII representation of a circuit. The aig format contains the same sections, but uses certain encoding and omitting to get a smaller sized file. The same information is still present for both. The sections contain in the following order:

1. The wire numbers for all the inputs
2. The wire numbers for all the latches including the input wire number and their initial values
3. The wire number of the bad state property
4. The wire numbers of the constraint
5. The wire numbers of the AND-gates followed by the wire numbers of the inputs

6. An symbol section that allows annotation of inputs, latches and AND-gates using their signal numbers

7. A comment section holding information about the circuit

The aig format differs slightly, with the input section being omitted and the AND-gate section being binary encoded.

3.3 Pre-existing tools

To manipulate, model check and certify witness circuits a set of tools is needed. The first tool needed is called AIGER [1]. This is a tool to manipulate aig files. The tool can be used to add input, latches or AND-gates or turn aig files into aag files. Included inside the repository of AIGER is a header file containing an API for the AIGER library. This API can be used to check the validity of an aig file and get useful information like the location of constraints or bad state properties. This API will also automatically update the header of the file if manipulations were done. To make sure a hardware design does not violate certain bad state properties, it is first run through a model checker. The model checker being used for this thesis is rIC3 [4]. This model checker uses IC3 to model check, which is a SAT-based model checking algorithm. The model checker works by giving a circuit that needs to be model checked and a file that will serve as the witness circuit. After calling rIC3, a short report is printed. This report will tell whether the property is satisfiable or not. If rIC3 reports that the property is unsatisfiable (UNSAT), then a witness circuit is produced. If the property is satisfiable, a trace showing how to satisfy the property is given. The model checking process works the same if a model is preprocessed, but this also requires the acquired witness circuit to be post processed. This difference is shown in figure 5.

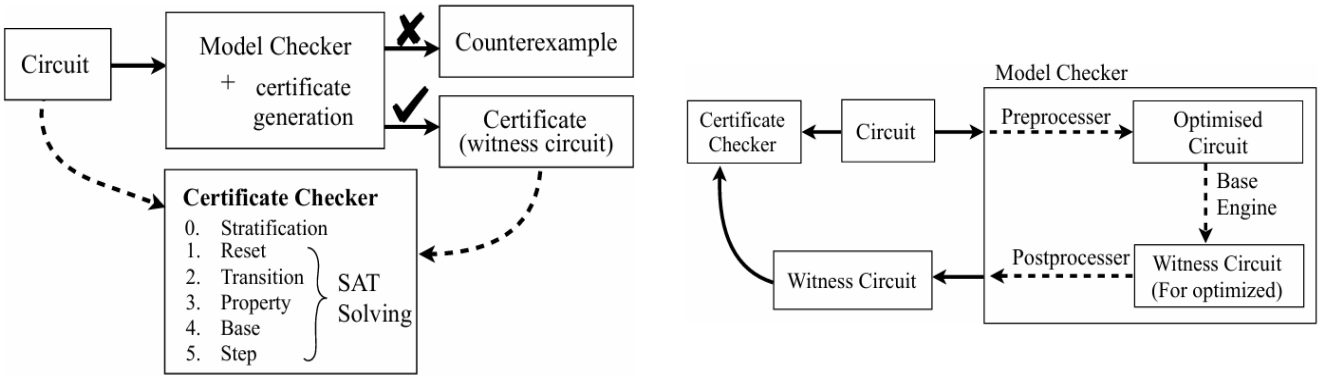

Figure 5: An overview of the model checking certification process (left), and the certification flow when preprocessing is employed in a model checker (right). [7]

Once a witness circuit is produced, its correctness needs to be checked. The reason for this is because a model checker may incorrectly produce a witness circuit that does not guarantee the safeness of a model. An independent certificate checker that only focuses on witness circuit correctness is necessary to make sure a safety property holds. For this thesis the certificate checker certifaiger [3] is used, because it focuses on circuits in the AIG format. The way certifaiger checks the validity of a witness circuit is by performing 5 checks. These checks are described in the formal definition of a witness circuit described in section 2. The last tool being used is called AIGMERGE. This tool is used to merge two witness circuits together. This is done in the way described in section 2. The tool takes a number of witness circuits and combine them to make a new one. This construction will later be used to construct witness circuits for certain hardware designs.

4 Approach

4.1 The code

To investigate the extracting and certifying of constraints, a tool is needed that can do both for a given model and it's properties. In this section, the approach of writing the code, implementing constraint extraction and certifying properties is described. This also includes the challenges that come with constraint extraction and the total functionality of the written code. A brief overview of the pipeline for the experiments in section 5 is given.
To start, the tool is written in C++. The tool needs a way to produce a witness circuit and to certify it. As mentioned before rIC3 is used for model checking, which includes producing a witness circuit. To certify this witness circuit, certifaiger is used. For this paper, a local build is made from both programs. This is done by downloading their code from the GitHub repositories and running the make commands to get a binary for both programs. The code calls these programs using a system call.
The AIGER API comes in the form of a header file. This header file is included in the code of the tool, which gives access to aig file manipulation functions.
To run experiments for the tool, fitting hardware designs are needed. The Hardware Model Checking Competition (HWMCC) [2] provides high level circuits that work great as benchmarks for the tool. Specifically, models that have multiple properties were chosen for the experiments. Each of these models has multiple files, where each file contains one of the properties. Each property is depicted by a number, as shown in figure 6.

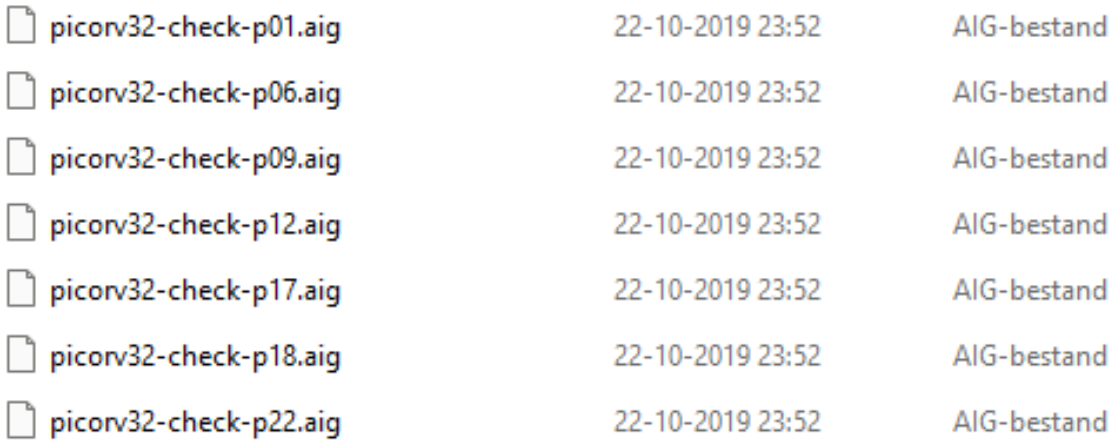

Figure 6: Example of how a model is split into files based on properties

4.2 Challenges

When building the tool, a few challenges need to be addressed. One of these challenges is the input/latch remapping rIC3 can produce when making a witness circuit. An example of such a remapping is shown in figure 7.

```
i278 = 924
i279 = 926
i280 = 928
i281 = 930
i282 = 932
i283 = 934
i284 = 936
l0 = 938
l1 = 940
l2 = 942
l3 = 944
l4 = 946
l5 = 948
l6 = 950
l7 = 952
l8 = 954
l9 = 956
l10 = 958
```

Figure 7: Example of input and latch remapping in an aag file.

The remapping produced involves input and latch signal numbers. When making a witness circuit, the model checker can decide to reassign the signal numbers of input and latches. This creates a discrepancy between the signal numbers in the model and in the witness circuit. That is why the remapping is added as a comment in the aag file of the witness circuit. The letter and number at the start of the line represent the input or latch of the model and the letter after the equal sign represents that same input or latch in the witness circuit.
The issue is that this remapping is not taken into account by AIGMERGE. When this remapping is not present, the certification does not work. Therefore, a separate function needed to be made that can extract this remapping and add it to the merged witness. This is implemented in the tool by using the standard template library of C++. By reading in the file and looking for the comment section and copying this section into the merged witness, the issue is fixed.
Another major issue is the making of copies of the models. When using a property as a model constraint, the original model needs to be modified. Since the original model is still needed for the certification step, a copy has to be made that gets the property added as constraint. The problem is how to keep track of all these files and where to store the created copies.
To keep track of the files, vectors are used. Before doing any model checking, the vectors are filled with the names of the relevant files in a certain order. This is to make sure that files that belong together appear in the same spot in all vectors.
the created copies are stored inside the folder that is given to the program for easy access. When

the program is done running, irrelevant files can then be easily removed.

4.3 Functionality

In the end, the tool is a compiled C++ script that uses command line inputs to run different tasks. These tasks are related to the different experiments that will be performed. The tool expects a folder that contains all the aig files that need to be model checked and certified.
The first functionality of the tool involves model checking and certifying individual circuits without constraint extraction. This is means the tool will run rIC3 and certifaiger for every single file and its corresponding witness circuit. Figure 8 demonstrates how for each circuit a single witness file is created.

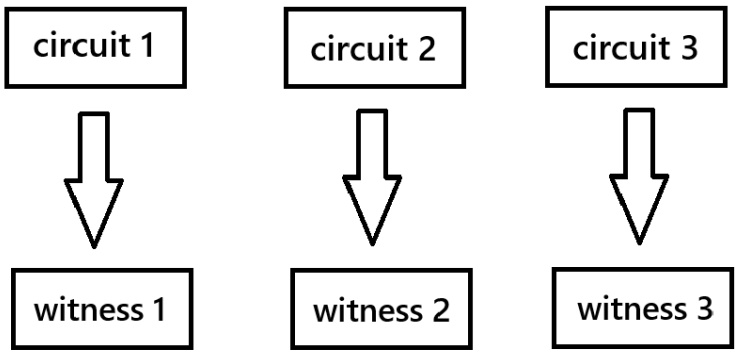

Figure 8: An example of how the standard functionality produces witness circuits.

The second functionality of the tool is to create witness circuits using constraint extraction.

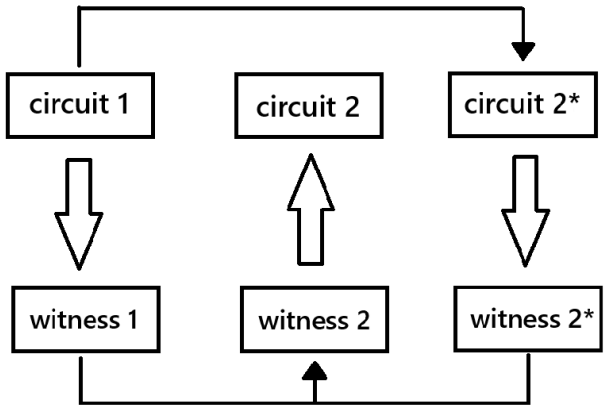

Figure 9: An example of how constraint extraction changes the functionality and creation of the tool.

To create a witness for a circuit, a copy is first made. This copy gets the property of the first circuit added as a constraint. Model checking will then give a witness for this copy. The witnesses for the model constraint and the copy are then merged to make a witness for the circuit. This workflow is shown in figure 9.
Besides these main functions, the tool can also be used to individually model check a circuit or to individually certify a certificate.

4.4 Experiment setup

For the experiments, benchmarks with multiple properties will be used. To measure time, the chromo library, which is a standard template library in C++, is being used. This provides and instruction that accurately measures the time when the instruction is executed. This function is used to measure the time difference using a steady clock, which is a monotonic clock that will never be adjusted. This difference will then be used to measure the time it took to execute a set off instructions in seconds as can be seen in section 5.
The machine that the experiments are run one has the following setup and specifications:

- AMD Ryzen 7 5800H
- 16,0 GB RAM
- Windows 11, using WSL 2
- The machine is plugged in

5 Experiments

After running the experiments, the results are shown in table 1.

	Total time normal	total time one constraint	Total time multiple constraints
picorv32 (7)	2,7429	2,8183	4,6945
zipcpu (8)	44,7032	28,5644	>1000
zippfcache (7)	3,3935	3,2835	6,9353
sm98 (2)	42,8631	41,8158	41,8591
mentor (10)	10,9971	11,4962	57,5659

Table 1: The total time (in seconds) for model checking and certifying a set of benchmarks. The amount of properties is shown in the brackets next to the benchmark name

In this table, the average time of completing the model check and certifying for both methods explained above are shown. In addition to this, a measurement is done which adds just one constraint to each circuit.
In figure 10 it shows the speedup gained in model checking depending on the amount of constraints added to a circuit. Note that a speedup below 1 means a decrease in speed. The figure shows that overall, adding constraints does not lead to a remarkable change in time to modelcheck circuits.

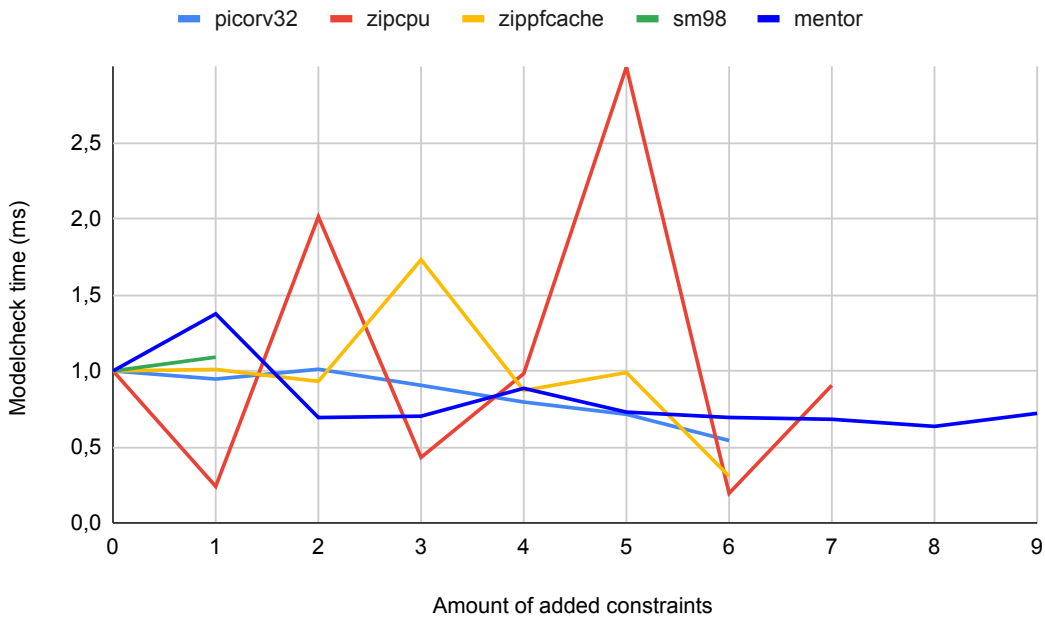

Figure 10: Chart showing the average time to model check a modified circuit.

Figure 11 shows the speedup in certifying a witness when using a merged witness. These merged witness were made by merging normal and previously merged witnesses. The chart shows a decrease in certifying time compared to certifying normal witnesses. It should be noted that by increasing

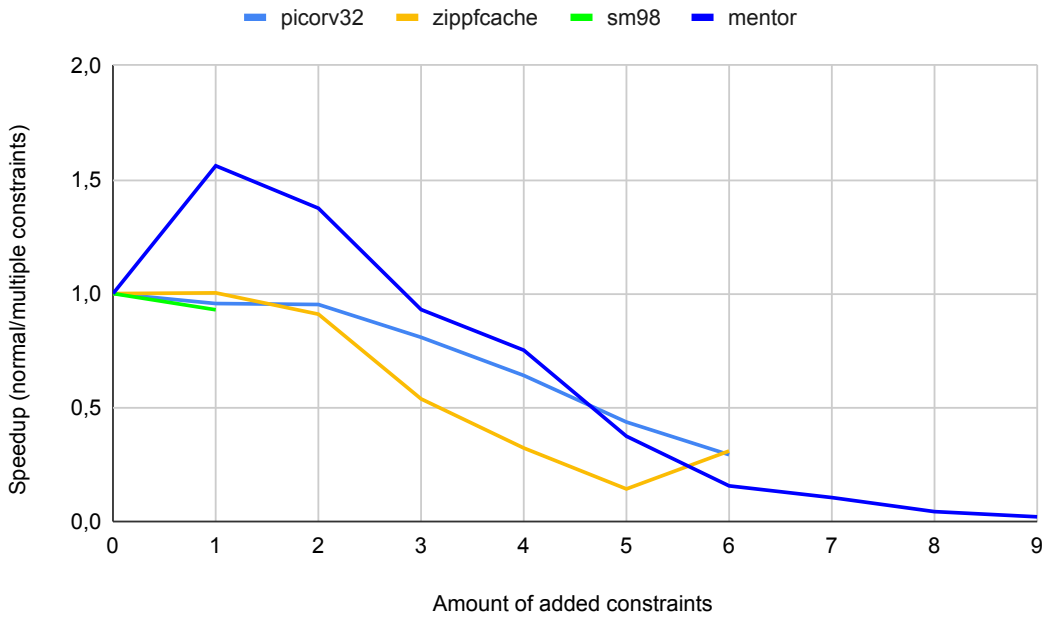

Figure 11: Chart showing the speedup obtained for certifying a merged witness.

the amount of constraints also mean increasing the size of the witness circuits.
To investigate the effect of using non merged circuits to merge with, a different construction for creating merged witnesses was used. In this construction, standard witnesses are created first for every circuit. After this, the merged witnesses for the circuits are created using these non-merged witnesses. The speedup in certifying time is shown in figure 12.
This chart shows a faster certifying time using these optimal witnesses. The reason this construction is not taken into account for the total time is because it requires to first model check every circuit before making the witnesses.

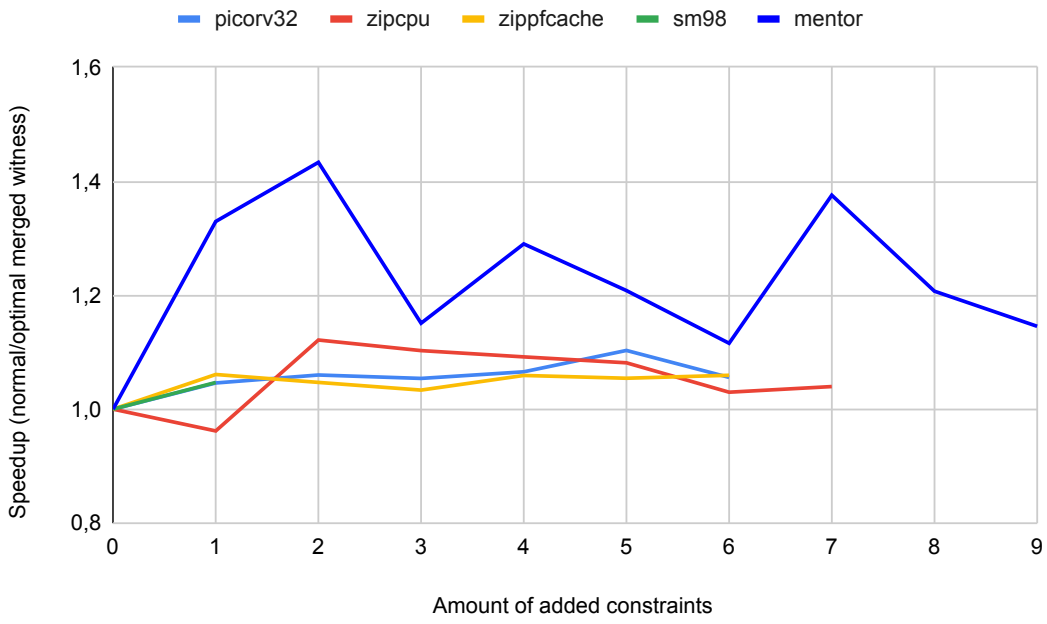

Figure 12: Chart showing the speedup obtained for certifying a merged witness that is created from non merged witnesses.

6 Conclusions

After observing the results from section 5, it can be stated that the use of constraint extraction and certification can increase model check and certifying time. When using the construction where witness circuits are created using other merged witnesses, the time to certify a witness increases. This becomes worse the more constraints are added. The reason for this is the increasing file size for the witness. This is the reason why the total time with using just one constraint is very effective. The witness for the one constraint is created through model checking, not by merging. this keeps the file size small and decreases the time to certify the witness. This is why the total time for this construction is lower then the standard one.

7 Future work

Currently, the most effective way to use the tool is to use the construction that uses only one constraint. For future work, two things can be considered. First is the use of different kinds of constraints. As mentioned in [7], there are also other constraints besides model constraints, such as property and inductive constraints, that can be certified. This would require suitable benchmarks that have these kind of constraints of course. Another thing to consider is a way to automatically extract these kinds of constraints. considering different techniques would have an impact on the efficency of model checking and certifying.

References

[1] Aiger tool. https://github.com/arminbiere/aiger. Accessed: 20-02-2026.

[2] Benchmarks from hardware model design competition. https://hwmcc.github.io/2025/. Accessed: 03-03-2026.

[3] Certifaiger tool. https://github.com/Froleyks/certifaiger. Accessed: 20-02-2026.

[4] model checker ric3 tool. https://github.com/gipsyh/rIC3. Accessed: 20-02-2026.

[5] Keijo Heljanko Armin Biere and Siert Wieringa. Aiger 1.9 and beyond. *Johannes Kepler Universität Linz, Institut für Formale Modelle und Verifikation.*

[6] M. Morris Mano and Charles R. Kime. *Logic and Computer Design Fundamentals 3th edition.* Pearson Education, 2008.

[7] Armin Biere Nils Froleyks, Emily Yu and Keijo Heljanko. Certifying constraints in hardware model checking. *International Symposium on Formal Methods (FM)*, 2026.

[8] Lucas Wagner, Alain Mebsout, Cesare Tinelli, Darren Cofer, and Konrad Slind. Qualification of a model checker for avionics software verification. In *NASA Formal Methods Symposium*, pages 404–419, 04 2017.