Zigzagging Strategies for Temporal Induction

Guillaume Baud-Berthier
SafeRiver
Montrouge, France
guillaume.baud-berthier@safe-river.com

Laurent Simon
Bordeaux-INP, LaBRI,
CNRS UMR 5800,
University of Bordeaux
France
lsimon@labri.fr

***Abstract*—Model Checking is at the heart of formal methods for software and hardware verification. In this area of active research, Bounded Model Checking (BMC) and k-induction have reached very impressive results, especially when both methods are working together. They are based on a common approach that unrolls the transition relation, but each method serves a different purpose in practice. BMC is usually used for bugs findings, while k-induction aims at building inductive invariants. The ZigZag approach, proposed 15 years ago, takes benefit from both strategies by successively calling each one of them, while trying to share a lot of information between calls thanks to the mechanism of SAT clauses learning.**

Despite the practical importance of the ZigZag algorithm, it was mainly used forwardly until last year. The transition relation was unrolled by increasing depths only. However, as stated by the authors of ZigZag themselves, it was possible to consider the ZigZag approach backwardly. The experimental study of backward zigzag performances was only proposed one year ago.

In this paper, we propose to extend the idea of the ZigZag algorithm by allowing to unroll the transitions from the middle. This has the nice property of allowing the SAT solver to keep learnt clauses that are both close to the initial state and to the bad state in the search. Our experimental study however shows that the best option for ZigZag is still to perform it backward, as stated in a previous work. However, we also show that our hybrid approach offers the same performances as forward ZigZag, while allowing more flexible strategies to be developed in the future, for example by choosing the right transition to expand.

I. Introduction

Model Checking (MC) is an important part in the design process of critical components (hardware and software). For practical reasons, it is often implemented by a Bounded Model Checking approach (BMC, [1]), that checks if there exists a fixed length execution path leading to a system state contradicting the given specification. This kind of problems is probably one of the first successful industrial application of SAT solvers. Yet, due to its bounded aspect, BMC is somehow oriented towards bugs finding, where bugs in software and hardware design are expected to be found within a limited bound. Hence, when the correctness of the system must be ensured, it may not be obvious to find the maximal length of the paths to check, or this length can be too large to be used in practice. Induction-based algorithms such as k-induction [2] allow to get rid of this sequential depth by building inductive invariant.

Partially supported by the French Research National Agency with the SATAS project ANR-15-CE40-0017

Both BMC and k-induction efficiency relies on their underlying SAT solver. These methods may generate huge SAT formulas, which can exceed several million clauses and variables, or even require thousands of SAT calls for solving a single instance. Historically, this kind of problems had a very important impact on the design of SAT algorithms used in practice. Conflict-Driven Clause Learning solvers (CDCL, [3], also called "modern" SAT solvers) were initially designed to be able to cope with this kind of very large instances.

In this context, the ZigZag algorithm [4] aims at taking the best of BMC and k-induction by successively calling each of them. However until very recently, Zigzagging was only performed forwardly, by following increasing depths. It was only demonstrated one year ago that performing Zigzag backward was a more efficient option [5]. In this paper, we propose to generalize the Zigzag algorithm by allowing to unroll transitions from the middle. We show that, despite a less trivial encoding, our approach obtains similar results than the forward Zigzag approach, while allowing more options to be investigated in the future, for instance by carefully choosing where to unroll the transition.

II. Preliminaries

The satisfiability (SAT) problem addresses the well-known NP-Complete question, expressed in propositional logic, whether a given formula is satisfiable, *i.e.* if there exists an assignment of the variables that satisfies the formula.

A propositional *formula* consists of a set of boolean variables, logical constants: $\top$ (true, 1), $\bot$ (false, 0), and logical connectives: $\neg$ (NOT), $\wedge$ (AND), $\vee$ (OR). A *literal* is a variable or its negation: x, $\neg x$. A *clause* is a disjunction of literals, *e.g.* $x \vee \neg y \vee z$. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, *e.g.* $(x \vee \neg y \vee z) \wedge (\neg x) \wedge (y \vee z)$. We can also write a clause as a set of literals, *e.g.* $\{x, \neg y, z\}$, and a CNF formula as a set of clauses, *e.g.* $\{\{x, \neg y, z\}, \{\neg x\}\}$. An *assignment* is a mapping of variables to Boolean values ($\top$ or $\bot$). An assignment satisfies a formula if the formula is evaluated to $\top$ by substituting variables by their assigned values.

A. SAT Solvers General Principles

SAT solvers are often described as a highly optimized black box solving the SAT problem. Most of SAT solvers take

a formula in CNF as an input and are able to provide an assignment of the variables if the formula is satisfiable.

With the introduction of CDCL (Conflict Driven Clause Learning) algorithms [3], [6], [7], learning clauses became the cornerstone method of SAT solvers. However, as CDCL solvers can learn more than 5,000 clauses a second, carefully managing the learnt clause database has quickly been crucial. In our approach, we use Glucose [8], a state of the art SAT solver that has the particularity of removing a lot of learnt clauses (on a typical run of Glucose, 95% of the learnt clauses could have been remove), thanks to a measure over the quality of clauses. What is important in our context is the ability of this SAT solvers to identify important clauses to keep from runs to run. We will indeed use a particular mode of operation for SAT solvers: incremental SAT (see [9] for a more detailed) exactly as it was defined in Minisat [7]. The idea is to be able to keep the same SAT engine from calls to calls (and with it all the interesting learnt clauses) by activating and deactivating some sets of clauses. It is the responsibility of the SAT solver to be able to handle which learnt clauses to keep and which learnt clauses to deactivate (in practice, this is done with *assumption literals*, see section II-B3).

B. Model Checking

Model checking is an automatic, exhaustive technique of formal verification checking whether a given *model* verifies a given *property*. Models are usually encoded as state machines and properties (expressing a particular behavior of the system) are generally given in temporal logic (for instance Boolean finite-state transition systems for the model and AGp (CTL* logic) for asserting properties).

Definition 1: A *finite-state transition system* is a tuple $\mathcal{M} = \langle V, I, T, P\rangle$, where V is a set of Boolean variables, $I(V)$ is a formula over V describing the initial states, $P(V)$ a formula over V describing the states verifying the property, and $T(V, V')$ is the transition relation, *i.e.* a formula over V,V' defining the accepted transitions. $V' = \{v' \mid v \in V\}$ is the primed version of the set V, usually used to represent next state variables.

By extension, we will use $V_i = \{v_i \mid v \in V\}$ to express the same sets as V with renamed variables when unrolling the transition relation. Assignments satisfying a formula over V represent states of the system, so we will often refer to formulas as sets of states. Thus, we define $Bad(V) = \neg P(V)$ as the set of states that contradict the desired behavior of the system. *Bad* (Resp. *safe*) states correspond to the set of states verifying $Bad(V)$ (resp. $P(V)$). A system admits a *counterexample* if it is possible to reach a bad state from the initial states by repeatedly applying the transition relation. Conversely, the property is verified if there is no path from the initial states to a bad states. In other words, the set of reachable states and the set of bad states are disjoint.

1) Bounded Model Checking: Bounded model checking is a parameterized model checking algorithm focusing on counterexamples detection [1]. Intuitively, the idea is to build a formula that is satisfiable if the model can reach a bad state in k transitions. BMC is very efficient for bug finding, but not used to prove properties in practice. Indeed, it is hard to find a suitable depth k for which the property is ensured to be verified[1]. Formally, BMC is defined as follow:

$$\text{BMC}_k = I(V_0) \wedge \bigwedge_{i=0}^{k-1} T(V_i, V_{i+1}) \wedge Bad(V_k)$$

In other words, the formula expresses that there exists a path of length k from the initial states to a bad state. This formula is usually encoded in CNF and its satisfiability is determined using a SAT solver. However, this definition does not allow to check BMC_k and claim that there is no counterexample in less than k transitions. Hence, model checkers usually ensure soundness by performing BMC incrementally, *i.e.* start with $k = 0$ and increase k by 1 after each solver call. Another less common approach consists in extending $Bad(V_k)$ to $\bigvee_{i=0}^{k} Bad(V_i)$.

2) k-induction: k-induction is a generalization of the simple induction principle. Simple induction proof consists in two parts: the base case and the induction step. The base case expresses that the initial states verify the property. The induction step states that the transition relation preserves the property, *i.e.* the set of states reachable in one transition from the safe states are safe states too. Formally, the two parts are defined as follow:

$$I(V) \Rightarrow P(V) \qquad \text{(base)}$$
$$P(V) \wedge T(V, V') \Rightarrow P(V') \qquad \text{(step)}$$

The base case checks if the initial states and the bad states are disjoint, while the induction step checks if a bad state can be reached in one transition from the safe states. Simple induction, equivalent to 1-induction, attempts to show that the property is an inductive invariant. However, in practice, properties are rarely inductive by themselves, meaning that induction step fails, *i.e.* is satisfiable. In this case, no conclusion can be drawn and the property must be strengthened. In k-induction, first introduced in [2] as *temporal induction*, the induction hypothesis is strengthened by a path of k consecutive time steps where the property is assumed to hold. Hence, the base case must be extend to ensure that the property is verified for the k first time steps. BMC_k is used as base case formula and the induction step formula k-ind is defined as:

$$k\text{-ind} = \bigwedge_{i=0}^{k-1} \left[P(V_i) \wedge T(V_i, V_{i+1})\right] \wedge Bad(V_k)$$

3) ZigZag Algorithm: ZigZag algorithm, introduced in [4], simply consists of combining BMC_k and k-induction inside a single solver. k-ind and BMC_k are performed consecutively for $k = 0, 1, \ldots \infty$, until either a BMC formula becomes satisfiable, *i.e.* there exists a counterexample, or a k-ind formula becomes unsatisfiable, the property is verified. For

[1]The number of model states is obviously a sufficient depth but the satisfiability checking of the generated formula is usually too hard to be solved.

clarity, in the following, we will use I_i, P_i to represent $I(V_i)$, $P(V_i)$ and $T_{i,j}$ in place of $T(V_i, V_j)$. Let us illustrate how ZigZag works on the first SAT calls:

$$\begin{array}{rl} \text{0-ind:} & \big[Bad_0\big] \\ \text{BMC}_0\text{:} & \big[I_0\big] \ \wedge \ \big[Bad_0\big] \\ \text{1-ind:} & P_0 \wedge T_{0,1} \wedge \big[Bad_1\big] \\ \text{BMC}_1\text{:} & \big[I_0\big] \ \wedge \ P_0 \wedge T_{0,1} \wedge \big[Bad_1\big] \\ \text{2-ind:} & P_0 \wedge T_{0,1} \wedge P_1 \wedge T_{1,2} \wedge \big[Bad_2\big] \\ \text{BMC}_2\text{:} & \big[I_0\big] \ \wedge \ P_0 \wedge T_{0,1} \wedge P_1 \wedge T_{1,2} \wedge \big[Bad_2\big] \end{array}$$

Square brackets are used to highlight the temporary formula parts. These parts are usually *activated/deactivated* with the help of assumption literals (see [7], [10].

An assumption literals is a literal that is added to a set of clauses in order to easily modify their trivial satisfiability, while leaving the rest of the formula untouched. For instance, suppose a solver S with clause $c = \{\neg act, x, y\}$ in which the assumption literal act was added. It is easy to activate c by forcing act to $\top$ in S. The solver has then also to satisfy the constraint $x \vee y$. On the contrary, by forcing act to $\bot$, c is deactivated, the solver may return a model with $x = \bot, y = \bot$.

In practice, by solving BMC$_k$ and k-ind formulas consecutively inside the same running instance of a solver, the solver can benefit from clauses deduced from the common part of both formulas, *i.e.* the unrolling of the transition relation and the parts related to the property. Our hypothesis is that it is important to allow the solver to keep good clauses from runs to runs.

4) Backward ZigZag: The ZigZag algorithm can be performed in two directions: forward or backward, as the authors of [4] well explained it. To illustrate the differences between backward and forward ZigZag, let us write the first SAT calls of the ZigZag algorithm, in backward:

$$\begin{array}{rr} \text{0-ind:} & Bad_0 \\ \text{BMC}_0\text{:} & \big[I_0\big] \wedge Bad_0 \\ \text{1-ind:} & P_1 \wedge T_{1,0} \wedge Bad_0 \\ \text{BMC}_1\text{:} & \big[I_1\big] \wedge P_1 \wedge T_{1,0} \wedge Bad_0 \\ \text{2-ind:} & P_2 \wedge T_{2,1} \wedge P_1 \wedge T_{1,0} \wedge Bad_0 \\ \text{BMC}_2\text{:} & \big[I_2\big] \wedge P_2 \wedge T_{2,1} \wedge P_1 \wedge T_{1,0} \wedge Bad_0 \end{array}$$

There are two main differences between the forward and the backward version. First, the part of the formula that stays activated at the same depth for BMC$_k$ and k-ind, *i.e.* the *static* part, is not the same: In the forward version, it corresponds to the initial states, whereas in the backward version it is the bad states. Second, as the initial states are only activated for BMC, this means that the static part is activated in every SAT calls in the backward version. Somehow surprisingly, it was shown in [5] that backward zigzag was more efficient than forward zigzag.

III. Hybrid ZigZag

In this section, we propose to generalize the notions of forward / backward Zigzag by proposing to unroll the transition relation from the middle.

As reported in [5], ZigZag seems to be more efficient for finding CEX when performed forward. At the opposite, it is generally faster to check models with a valid property when the transition relation is performed backward. This observation makes sense when observing which part of the formula stays static between SAT calls (we will make the same observation in the experimental part, see figure 2 (top sub-figure)) It can be noticed here that SAT solvers are used in an incremental way, and thus good learnt clauses can be kept from calls to calls, if they depend only on the current static part of the formula. For instance, a learnt clause derived by resolutions from clauses belonging to the $Bad_0 \wedge T_{1,0} \wedge P_1 \wedge T_{2,1} \wedge P_2$ (see lines 2-ind and BMC_2 in the backward ZigZag example) will be kept from runs to runs until the end. At the opposite, clauses built by resolutions from clauses of $P_0 \wedge T_{0,1} \wedge P_1 \wedge T_{1,2}$ (see lines 2-ind and BMC_2 in the forward ZigZag example) can be kept in the forward ZigZag scenario. We see here how important the unrolling direction is important for preserving good learnt clauses.

We propose to generalize this observation by allowing clauses close to the initial property and clauses close to the bad states to stay in memory. Our idea is simply to unroll half of the transitions forward and half of the transitions backward, by alternatively selecting which direction to unroll. For simplicity, let us partition the formula into 3 parts. Let us first define the part encoding the forward unrolling:

$$T_{av}(k) \ := \bigwedge_{i=0}^{(k-1)/2} \big(P_{2i} \wedge T_{2i,2i+2}\big)$$

Then, let us define the backward unrolling:

$$T_{ar}(k) \ := \bigwedge_{i=0}^{(k-2)/2} \big(P_{2i+3} \wedge T_{2i+3,2i+1}\big)$$

We need now to define the part of the formula that connects the two directions. The only variables required to connect two distinct depths are the latches variables. Hence, we will use m to state the subset of V representing the latch variables only.

$$EQ(k) \ := m_{2\cdot\lfloor\frac{k+1}{2}\rfloor} = m_{2\cdot\lceil\frac{k+1}{2}\rceil-1}$$

Thus, for a given k, the hybrid unrolling formula is defined as follows:

$$\big[I_0\big] \wedge T_{av}(k) \wedge \big[EQ(k)\big] \wedge T_{ar}(k) \wedge Bad_1$$

Parts of the formula that are in brackets are subformulas that can be activated or deactivated thanks to assumption literals during SAT calls. I_0 is only activated for BMC calls (and thus deactivated for k-induction calls). $EQ(k)$ constraints are activated and deactivated following unrolling steps. When the ZigZag algorithm is at depth k, the constraint $EQ(k)$ is activated and all the other $EQ()$ constraints are deactivated (e.g. $EQ(0)$ is only activated for solving formulas BMC$_0$ and 0-ind). A more detailed example of how our hybrid ZigZag

$$
\begin{aligned}
\text{0-ind:} &\quad [m_0 = m_1] \wedge Bad_1 \\
\text{BMC}_0\text{:} &\quad \big[I_0\big] \wedge [m_0 = m_1] \wedge Bad_1 \\
\text{1-ind:} &\quad P_0 \wedge T_{0,2} \wedge [m_2 = m_1] \wedge Bad_1 \\
\text{BMC}_1\text{:} &\quad \big[I_0\big] \wedge P_0 \wedge T_{0,2} \wedge [m_2 = m_1] \wedge Bad_1 \\
\text{2-ind:} &\quad P_0 \wedge T_{0,2} \wedge [m_2 = m_3] \wedge P_3 \wedge T_{3,1} \wedge Bad_1 \\
\text{BMC}_2\text{:} &\quad \big[I_0\big] \wedge P_0 \wedge T_{0,2} \wedge [m_2 = m_3] \wedge P_3 \wedge T_{3,1} \wedge Bad_1 \\
\text{3-ind:} &\quad P_0 \wedge T_{0,2} \wedge P_2 \wedge T_{2,4} \wedge [m_4 = m_3] \wedge P_3 \wedge T_{3,1} \wedge Bad_1 \\
\text{BMC}_3\text{:} &\quad \big[I_0\big] \wedge P_0 \wedge T_{0,2} \wedge P_2 \wedge T_{2,4} \wedge [m_4 = m_3] \wedge P_3 \wedge T_{3,1} \wedge Bad_1 \\
\text{4-ind:} &\quad P_0 \wedge T_{0,2} \wedge P_2 \wedge T_{2,4} \wedge [m_4 = m_5] \wedge P_5 \wedge T_{5,3} \wedge P_3 \wedge T_{3,1} \wedge Bad_1
\end{aligned}
$$

Fig. 1. Illustration of unrolling steps for our Hybrid ZigZag algorithm

works is illustrated figure 1. As we can see, all the learnt clauses implying forward unrolling close to the initial statue can be kept if they are good enough for the solver to decide so. Good clauses involving backward transitions and close to the bad stats can also be kept. However, it is unknown whether the cost of connecting both formulas thanks to the $EQ(k)$ constraints is prohibitive or not. This is addressed in the following section.

IV. Experimental Results

In order to compare the performances of the different strategies for ZigZag, we implemented a tool working on standard format AIGER [11]. Our tool had a builtin preprocessor for AIG models, providing usual features, s.t. constants propagation, rewriting rules (idempotency, contradiction, subsumption, etc.), structural hashing, cone of influence. No sweeping was performed. Moreover, variables elimination [12] was only performed at model level following a method somewhat analogous to the one in ABC [13]. Also, our ZigZag algorithm (forward, backward and hybrid) allowed the generation of simple path constraints on demand [4].

We used a slightly modified version of Glucose [14] as our SAT decision procedure. We ran our set of strategies on 547 benchmarks from the 2015 hardware model checking competition (HWMCC 2015). We set the time limit to 1 hour, and the memory limit to 8 GB. In some cases (when specified), we will consider in the remaining only results of *interesting* problems, *e.g.* problems taking more than one minute to solve.

We ran a set of 4 strategies: *Fwd*: Transitions are unrolled forward. *Bwd(1)*: Transitions are unrolled backward where constraints are systematically added to connect contiguous depths. *Bwd(2)*: Transitions are also unrolled backward but we used an encoding trick, *i.e.* we perform alias propagation to only add constraints when we don't have any other choice. *Hyb.*: Transitions are unrolled both forward and backward as described in the previous section.

Our first results are reported table I. From this table, it is clear that the method of choice is Bwd(2). As stated before, this result is not new. However, we would like to focus on the last column. The hybrid results offer the same results as the Fwd version, which is really encouraging given the number of constraints added by $EQ(k)$ to consider.

TABLE I
Forward (Fwd) vs Backward (Bwd(1-2)) vs Hybrid (Hyb.) ZigZag performance. Number of unsolved instances (UNK), solved instances with counter example (CEX) and number of solved instances for which the property was verified (PRO).

	Fwd	Bwd(1)	Bwd(2)	Hyb.
UNK	332	329	**327**	332
CEX	**119**	118	118	118
PRO	96	100	**102**	97

Let us now study the results in more details. We compare the results, figures 2, of all our versions one against the other. We restricted the points to all the problems that needed more than 10s to be solved by the 3 techniques (Fwd, Bwd(2) and Hyb). We observe a clear improvement when performing backward unrolling. As it can be seen figure 2 (top sub-figure) the backward approach seems to be even more efficient for the hardest problems, especially for PRO problems.

Results for our new hybrid approach are more mitigated. One more PRO instance is solved but one less CEX is solved. We can observe figure 2 (middle sub-figure) that, once again, the hybrid approach solves the same number of problems as the forward method, but is also slightly more efficient for the hardest problems.

However, when we compare the hybrid approach with the backward (Bwd(2)) approach, the later is clearly more efficient. However, if we omit the PRO results on the figure 2 (bottom sub-figures) results in terms of time are really comparable.

The question is now whether it could be interesting to add the hybrid method in a formal verification portfolio. The table II shows that there are some cases where unrolling the transition relation from the middle that can pay a lot. Of course, we carefully selected those benchmarks and it is clearly not always the case but we would like to point out that, as reported in the table's caption, in 25% of the cases, the hybrid approach was better than Fwd, Bwd(1) and Bwd(2) which is a very encouraging result at this stage of our investigations.

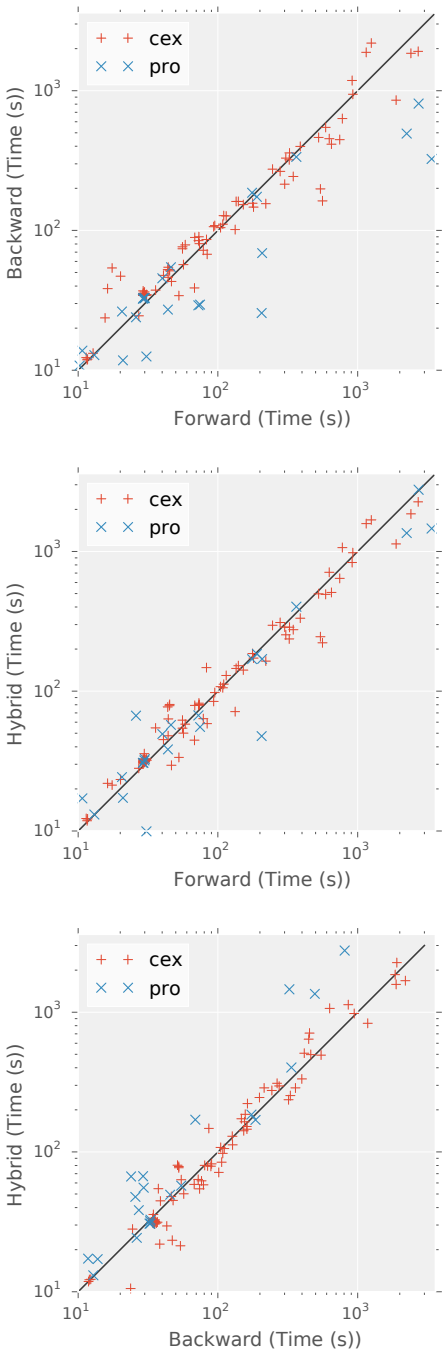

Fig. 2. Comparison of the time required to solve the same problems between the different strategies (forward, backward, and hybrid unrolling).

TABLE II
A SELECTION OF SOME GOOD CASES FOR OUR HYBRID METHOD. WE COUNTED 20 SIMILAR CASES (HYBRID IS FASTER THAN FORWARD AND BACKWARD 2) OVER THE 78 CASES WHERE ALL ZIGZAG VERSIONS NEEDED MORE THAN 10S TO GIVE AN ANSWER. TIME IS GIVEN IN SECONDS. THE MAXIMAL DEPTH REACHED IS GIVEN IN PARENTHESIS.

Instance	Fwd	Bwd	Bwd(2)	Hybrid
oski15a08b07	325 (11)	358 (11)	358 (11)	287 (11)
6s20	590 (8)	536 (8)	546 (8)	493 (8)
oski3b0i	911 (31)	950 (31)	1181 (31)	833 (31)
6s7	3447 (54)	T.O. (51)	T.O. (51)	2990 (54)

V. CONCLUSION

With the new unrolling process for the ZigZag method we introduced in this paper, one can easily imagine an adaptable algorithm (an heuristic) that choose to unroll the transition for the next depth, depending on where the SAT solver was working on, for example by analyzing where the more important conflicts occurred during the search (in terms of depths). Another interesting possibility is to check whether the solver is spending more time resolving BMC rather than the k-induction formulas. We could then chose to unroll the next transition from the initial state side rather than the bad states side and the other way around.

Even if our results are still below the performances of backward Zigzag, that was demonstrated only very recently, we showed that we obtain very similar results than forward Zigzag. We also observed a number of problem where the results are very promising while we have to consider a lot of additional constraints. This latter issue might be an interesting point for future work: for instance it could be possible to reduce the number of these constraints with a better encoding.

REFERENCES

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 1999, pp. 193–207.
[2] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *International conference on formal methods in computer-aided design*. Springer, 2000, pp. 127–144.
[3] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.
[4] N. Eén and N. Sörensson, "Temporal induction by incremental sat solving," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 4, pp. 543–560, 2003.
[5] G. Baud-Berthier and L. Simon, "On selecting constraints for replication in model checking," in *International Conference on Tools for Artificial Intelligence (ICTAI)*, 2017.
[6] J. P. M. Silva and K. A. Sakallah, "Graspa new search algorithm for satisfiability," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society, 1997, pp. 220–227.
[7] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.
[8] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers." in *IJCAI*, vol. 9, 2009, pp. 399–404.
[9] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.
[10] G. Audemard, J.-M. Lagniez, and L. Simon, "Improving glucose for incremental sat solving with assumption: Application to mus extraction," in *Proceedings of SAT 2013*, 2013.
[11] A. Biere, "The aiger and-inverter graph (aig) format," *Available at fmv. jku. at/aiger*, 2007.
[12] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *International conference on theory and applications of satisfiability testing*. Springer, 2005, pp. 61–75.
[13] N. Eén, A. Mishchenko, and N. Sörensson, "Applying logic synthesis for speeding up sat," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2007, pp. 272–286.
[14] G. Audemard and L. Simon, "The glucose sat solver," 2013.