Core Minimization in SAT-based Abstraction

Anton Belov* Huan Chen* Alan Mishchenko† Joao Marques-Silva*‡

* Complex and Adaptive Systems Laboratory, University College Dublin
Email: {anton.belov, huan.chen, jpms}@ucd.ie
† Department of EECS, University of California, Berkeley
Email: alanmi@eecs.berkeley.edu
‡ IST/INESC-ID Technical University of Lisbon

***Abstract*—Automatic abstraction is an important component of modern formal verification flows. A number of effective SAT-based automatic abstraction methods use unsatisfiable cores to guide the construction of abstractions. In this paper we analyze the impact of unsatisfiable core minimization, using state-of-the-art algorithms for the computation of minimally unsatisfiable subformulas (MUSes), on the effectiveness of a hybrid (counterexample-based and proof-based) abstraction engine. We demonstrate empirically that core minimization can lead to a significant reduction in the total verification time, particularly on difficult testcases. However, the resulting abstractions are not necessarily smaller. We notice that by varying the minimization effort the abstraction size can be controlled in a non-trivial manner. Based on this observation, we achieve a further reduction in the total verification time.**

I. Introduction

Abstraction engines are essential for ensuring the scalability of modern formal verification flows. A number of SAT-based abstraction methods rely on the capability of SAT solvers to derive unsatisfiable cores — the subsets of unsatisfiable CNF formulas that are sufficient to establish their inconsistency. Cores are used in proof-based abstraction (PBA) engines, introduced by Gupta et al. [1] and McMillan and Amla [2], to focus the abstraction process on the parts of the design that are (heuristically) relevant to the proof of the property under verification, and remove those parts that are known to be irrelevant, thus shrinking the abstraction size. However, while unsatisfiable cores are typically indeed significantly smaller than the formulas they are derived from, they may still contain many redundant clauses. Detection and removal of redundant clauses from unsatisfiable CNF formulas is essential reason for algorithms for computation of minimally unsatisfiable subformulas (MUSes) — an MUS is an unsatisfiable subformula that is minimal in the sense that removing any of its clauses makes it satisfiable. In recent years, a number of efficient algorithms for computing MUS have emerged (see [3] for a survey, and [4], [5] for recent developments). While the recently developed algorithms can handle very large problems successfully, computation of MUSes can still be a resource-intensive process — the related problem of testing the minimal unsatisfiability of propositional formula is complete for the complexity class DP[1][6], and the current algorithms compute MUSes using iterative calls to a SAT solver. However, given that the size and the quality of abstraction typically has a very significant impact on the success of the downstream verification flow, the computational resources spent on MUS-based core minimization might be offset and superseded by the subsequent gains. While the idea of using core minimization procedures in abstraction is not new, to our knowledge it has not been thoroughly evaluated.

In this paper we present the results of the first, to our knowledge, detailed study of the effects of core minimization using MUS in SAT-based abstraction algorithms. Specifically, we analyze the impact of the MUS-based minimization on a hybrid (counterexample- and proof-based) abstraction engine GLA (Gate Level Abstraction) [7] implemented in the publicly available verification framework ABC [8]. GLA is a highly optimized version of the single-instance SAT-based hybrid abstraction algorithm proposed in [9], although geared towards gate-level abstraction. We demonstrate empirically that, indeed, MUS-based core minimization can have a very significant impact on the size of the produced abstractions, and furthermore, that the computational overhead from minimization is often offset by the gains from the reduction in the subsequent proof time. Curiously, we also find that core minimization can at times result in *larger* abstractions. We investigate this issue further, and arrive at the conclusion that in order to get smaller abstractions, minimization has to be applied judiciously. We present the results of our study on a large set of industrial testcases from HWMCC'11 [10], and demonstrate that MUS-based minimization can result in over 5x reduction in the total verification time on a number of particularly difficult testcases.

II. Related work

Core minimization procedure based on re-running SAT solver with different parameters has been proposed by Gupta et al. [1] in the context of iterative SAT-based abstraction framework. In the same work the authors propose a heuristic to reduce the number of latches in the abstraction. As opposed to the work presented in this paper, neither of these procedures produce minimal cores. The work of Nadel [11] discusses a number of applications of MUS extraction in formal verification, one of which is in proof-based abstraction. However, the impact of MUS-based core minimization on the resulting abstraction size and quality (in terms of the proof time) has not been investigated in this work. Furthermore, in this paper, we use MUS-based minimization in a hybrid (counterexample and

978-3-9815370-0-0/DATE13/© 2013 EDAA

[1] DP is the set of languages L for which there are two languages $L_1 \in \mathsf{NP}$, $L_2 \in \mathsf{coNP}$ such that $L = L_1 \cap L_2$.

proof-based) abstraction framework — in Section V we argue that, in fact, hybrid abstraction frameworks are better suited for MUS-based core minimization than purely proof-based.

III. Background

In this paper we focus on model-checking safety properties of synchronous, single-clock hardware designs. To simplify the discussion, we assume that the design and a single safety property are given as a sequential And-Inverter-Graph (AIG) with a designated output gate p that represents the negation of the property — that is, the property is violated when p gets value 1. We will view such AIG as a set of (input, output, AND, flop) gates A. By $BMC(A,k)$ we will refer to the CNF formula obtained by unrolling A for k time frames (as, for example, in [12]), and asserting the value of the output gate p at frame k to 1. Thus, $BMC(A,k)$ is unsatisfiable if and only if $\mathcal{A}$ has no counterexamples of length k. By $cls(g,i)$, $0 \leq i \leq k$ we will refer to the set of clauses that correspond to the representation of the gate $g \in A$ at time frame i in the formula $BMC(A,k)$. Given a CNF $\mathcal{F} \subseteq BMC(A,k)$, we will say that a gate $g \in A$ is *included in* $\mathcal{F}$ if for some $0 \leq i \leq k$, $\mathcal{F} \cap cls(g,i) \neq \emptyset$.

A. Abstraction

Let A' be any subset of gates of A that includes the output gate p — the output signals of the gates from $A \setminus A'$ are replaced by *pseudo*-primary inputs. Then, A' constitutes a *conservative* abstraction of A — that is, A' preserves all behaviours of A, and so if the property p holds in A' it also holds in A. We will say that conservative abstraction A' of A is *precise to frame* k if A' has no counterexample of length k or less. Methods that construct conservative abstractions by removing logic that is deemed irrelevant to the proof of the property are referred to as *localization* abstraction methods. A number of methods for construction of localization abstractions have been proposed. Following [13], SAT-based abstraction methods can be classified into proof-based [1], [2] (those that use unsatisfiable cores to remove logic unnecessary to the proof), counterexample-based [14] (that build abstractions by adding logic to refute counterexamples), and hybrid counterexample- and proof-based [15], [9] (that alternate proof-based and counterexample-based stages). Note that in counterexample-based abstraction (CBA) the abstraction is built ground-up, starting from an empty set of gates, while PBA works in the opposite way — it starts from the full design, and removes logic, resulting in successively smaller abstractions. Additionally, abstraction methods are classified as gate-based vs latch-based, depending on whether the abstraction constructed from individual gates, or the latches together with their entire fanin cones.

B. MUS extraction

A CNF formula $\mathcal{F}$ is *minimally unsatisfiable* if (i) $\mathcal{F} \in$ UNSAT, and (ii) for any clause $C \in \mathcal{F}$, $\mathcal{F} \setminus \{C\} \in$ SAT. A CNF formula $\mathcal{F}'$ is a *minimally unsatisfiable subformula (MUS)* of a formula $\mathcal{F}$ if $\mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{F}'$ is minimally unsatisfiable. In general, a given unsatisfiable formula $\mathcal{F}$ may have more than one MUS.

Motivated by several applications (including PBA), minimal unsatisfiability and related concepts have been extended to CNF formulas where clauses are partitioned into disjoint sets called *groups* [16], [11].

Definition 1: Given an explicitly partitioned unsatisfiable CNF formula $\mathcal{F} = \mathcal{G}_0 \cup \cdots \cup \mathcal{G}_n$ (a *group-CNF formula*), a *group oriented MUS* (or, *group-MUS*) of $\mathcal{F}$ is a subset $\mathcal{F}' = \mathcal{G}_0 \cup \mathcal{G}_{i_1} \cup \cdots \cup \mathcal{G}_{i_k}$ of $\mathcal{F}$ such that $\mathcal{F}'$ is unsatisfiable and, for every $1 \leq j \leq k$, $\mathcal{F}' \setminus \mathcal{G}_{i_j}$ is satisfiable.
Note the special role of $\mathcal{G}_o$ (group-0) — the clauses of this group are always included in the GMUS. In a way, the set of groups $\mathcal{G}_1, \ldots, \mathcal{G}_n$ is minimized *with respect* to $\mathcal{G}_0$.

Over the years a number of effective MUS and GMUS computation algorithms have been developed. At the moment, the most efficient algorithms are based on a variation of the deletion-based approach, whereby clauses (resp. groups) are removed one at a time and the resulting formula is tested for satisfiability. If the formula is satisfiable, the clause (resp. group) is *necessary* and is included in the computed MUS (resp. GMUS). Some of the additional necessary clauses (resp. groups) can be detected using a technique called *model rotation* [17]. If the formula is unsatisfiable, the clause (resp. group) is removed, together with any other clause (resp. group) that does not appear in the unsatisfiable core returned by the SAT solver. See [4] for an example state-of-the-art MUS extraction algorithm.

IV. GLA— Gate-Level, hybrid Abstraction

In this paper we work with a gate-level version of the SAT-based *hybrid* abstraction algorithm developed in [9]. A number of important optimizations to this algorithm have been recently developed and implemented in the publicly available version of the verification framework ABC [8] (&gla command in ABC). The details of the optimizations are described in [7].

Given a sequential AIG A, the algorithm starts with the abstraction A' that includes only the output gate p of A. Beginning at frame $k = 0$, the algorithm adds gates to A' as long as the formula $BMC(A', k = 0)$ is satisfiable, i.e. as long as A' has counterexamples of length 0. If no gates can be added, the property is violated in frame 0, and the algorithm terminates. Otherwise, when $BMC(A', 0)$ becomes unsatisfiable, A' is precise to frame 0. However, given that the gates of A' that do not appear in the unsatisfiable core of the formula (returned by the underlying SAT solver) are not needed to prove the absence of counter-examples of length 0, these gates can removed from A' resulting in a smaller abstraction, that is still precise to frame 0. On the next iteration, the frame number k is incremented, and the algorithm proceeds to eliminate all counterexamples of length $k = 1$, by adding gates to A' until the formula $BMC(A', k = 1)$ is unsatisfiable. Once again, the unsatisfiable core of $BMC(A', k = 1)$ is analyzed to remove some of the gates from A' — note that this time care must be taken to remove only those gates (outside of the core) that do not appear in the abstraction A' computed in the previous iteration. The high-level flow of the algorithm is presented in Alg. 1. Here the abstraction A_{tmp} is used so that the algorithm has access to the abstraction A' computed in the previous iteration. The function `Refine-Abstraction`(A_{tmp}, A, τ, k)

Alg. 1: GLA(m) — Gate-Level Abstraction (with core minimization)

Input : Sequential AIG A, frame number k
Output: CEX or abstraction $A' \subseteq A$ precise to frame k

1 $A' \leftarrow \{p\}$ `// the output gate only, initially`
2 $k' \leftarrow 0$ `// time-frame number,` $0 \leq k' \leq k$
3 **while** $k' \leq k$ **do**
4 $\quad A_{tmp} \leftarrow A'$
5 $\quad$ **while** $BMC(A_{tmp}, k') \in$ SAT **do**
6 $\quad\quad \tau \leftarrow$ satisfying assignment of $BMC(A_{tmp}, k')$
7 $\quad\quad G_a \leftarrow$ `Refine-Abstraction`(A_{tmp}, A, τ, k)
8 $\quad\quad$ **if** $G_a = \emptyset$ **then return** CEX
9 $\quad\quad A_{tmp} \leftarrow A_{tmp} \cup G_a$
$\quad$ `//` $BMC(A_{tmp}, k') \in$ UNSAT `here`
10 $\quad G_r \leftarrow$ gates incl. in unsat. core of $BMC(A_{tmp}, k')$
MIN $\quad G_r \leftarrow$ `Minimize`(G_r, A', k') `// minimization`
11 $\quad A' \leftarrow A' \cup G_r$ `// Inv:` A' `is precise to` k'
12 $\quad k' \leftarrow k' + 1$
return A' `//` A' `is precise to` k

returns a set of gates whose addition to A_{tmp} will eliminate the counterexample represented by assignment τ. If the returned set is empty, the counterexample cannot be eliminated, i.e. it is a true counterexample. The details of the refinement algorithm are not relevant for this paper. However, it is worth to mention that in the implementation of GLA in ABC the refinement is performed using a priority-based refinement scheme introduced in [13].

The loop invariant (stated on line 11) guarantees that when the algorithm terminates and returns abstraction A', the abstraction is precise to frame k. In the typical verification flow, the abstraction is passed to a complete proof engine, which either determines that the property holds in A' (and, thus, in A), or finds a counterexample of length $> k$. If the counterexample is spurious (i.e. does not hold in A), one possibility is to re-run the abstraction algorithm with a higher frame bound, and initialized with A'.

One of the key advantages of the algorithm is that, as demonstrated in [9], it can be implemented using a single instance of an incremental SAT solver. This also makes it particularly suitable for the integration of MUS-based core minimization procedure, as described below.

V. Core Minimization in Abstraction

Earlier work that suggested to use MUS-based minimization in the context of PBA [11], has proposed to extract group-MUS from the BMC unrolling of the final abstraction. Here we are interested in "on-line" core minimization — that is, the goal is to shrink the current abstraction at the end of each major iteration of the algorithm Alg. 1. We note that hybrid abstraction algorithms (as opposed to PBA-based approaches) are particularly well suited for the application of MUS-based core minimization. This is due to the fact that abstractions are constructed ground-up and incrementally, and so the typical minimization problems that arise in this context are not too difficult for MUS extractors.

The function `Minimize`(G_r, A', k') on line MIN of Alg. 1 constructs a group-CNF instance from the current BMC formula, and runs a group-MUS extractor with the goal to reduce the number of gates added during the last refinement. The encoding to group-CNF is performed in the following way. Since the algorithm grows the abstraction monotonically, all gates that are included in the abstraction A' (from the previous time frame) must be included in the new abstraction. As such, the gates in A' are not subject to minimization, and so the image of these gates over the time-frames $0, \ldots, k'$ is added to group 0 (recall, this is the group that contains background clauses with respect to which the group-MUS computation is performed). Additionally, the property assertion p for frame k' is inserted into group 0. The gates from the set G_r that are not in A' are the new gates, added by the refinement procedure, and these are the gates on which we perform minimization. For each such gate g, we create a group $\mathcal{G}_g$ that contains the clauses that represent g in *all* time frames from 0 to k', that is $\mathcal{G}_g = \bigcup_{0 \leq i \leq k'} cls(g, i)$. The resulting group-CNF instance is then passed to a group-MUS extraction algorithm (we will describe the implementation details below), and the gates that correspond to the groups not included in the computed group-MUS are removed from G_r. Note that by the definition of group-MUS, the resulting set of groups, together with group 0, is unsatisfiable, and so the invariant of Alg. 1 still holds.

The fact that the final set of gates included in the abstraction is selected by the core minimization procedure (as opposed to being constructed by the SAT solver during the refutation of the BMC formula) allows a fine-grained control over the gates that are likely to end up in the abstraction. For once, group-MUS extractor may vary the order in which the groups are tested for necessity — groups that are tested first, and so the corresponding gates, are more likely to be removed. For example, we can force group-MUS extractor to first try to remove groups with a smaller total clause length — in the setting of GLA, this corresponds to removing flops, and thus prioritizing AND gates (keeping them in the abstraction). The reverse order would prioritize flops instead. Finally, we can force the extractor to keep particular gates by putting them into group-0. Additional possibility is to minimize the full set of gates in $A_{tmp} \setminus A'$, rather than the core G_r returned by the SAT solver. This is likely to cause additional overhead due to the minimization. However it might also give the extractor's heuristics more freedom to prioritize certain, "good", gates. Finally, we can also control the time limit for each call to `Minimize` — when the time limit is reached, a group-MUS over-approximation is returned to the main algorithm. Although we have implemented these fine-grained control options in our prototype (discussed below), in this paper we mostly deal with the results of a more high-level control of the *minimization effort* — this refers to making the decision on whether or not to apply minimization on a particular timeframe, or a range of timeframes. For example, minimization can be applied on every frame, or only on the first half of the frames, or only on the last 5 frames. As we found out, the minimization effort has a very peculiar, and somewhat unexpected, impact on the size of the resulting

abstractions. We will come back to this point in Section VII.

VI. Experimental study

A. Implementation details

We built a prototype implementation of GLAm by integrating an open-source MUS extractor MUSer2 [18] into the optimized implementation of GLA available in the public version of the verification framework ABC [8]. The MUS extractor in this prototype uses its own, separate, instance of a SAT solver, that is re-initialized in every invocation of `Minimize` in GLAm. Clearly this causes a large performance penalty since the SAT solver inside MUSer2 has to re-prove the instance on every call. To perform an objective comparison we forced MUSer2 to perform an initial call to the SAT solver with the instance prior to removing any groups, and discounted the time of this call. In the final version of GLAm the extraction will be performed using the SAT solver employed in GLA for BMC queries and refinement. This can be achieved, for example, by adding the activation literals to the clauses that correspond to the gates added during refinement, minimizing the core using the activation literals (as it is done in MUSer2), and asserting the activation literals for the removed and the remaining clauses correspondingly. Given that the number of refinement gates is typically relatively small, the overhead from activation literals is unlikely to be significant.

B. Experimental set-up and methodology

The set of benchmark testcases for our experiments was drawn from the set of single-property unsatisfiable (i.e. correct) benchmarks from HWMCC'11. The purpose of abstraction is to help verify properties that cannot be proved, or are very difficult to prove, on the full design. As such, to evaluate the effects of minimization in the context of abstraction, we selected those instances from the set that could not be proven with a modern proof engine within a non-trivial timeout. The proof engine for our experiments was chosen to be PDR [19] (also available in ABC) — a highly-optimized implementation of the powerful model-checking algorithm IC3 [20]. The instances that could not be proven in under 900 seconds were selected for our experiments.

As the ultimate goal of abstraction is to simplify property verification, the quality of abstraction should be associated not only with its size, but also with the effort (time) required to prove the correctness of the abstracted design. The following experiments focus on the evaluation of the quality of abstractions: to prove a property, an abstraction with higher quality requires shorter run-times. For each instance, the following steps are performed until the property is proved: starting from abstraction depth $k = 1$, (i) compute abstraction to depth k using GLA or GLAm (ABC command `&gla` with the default parameters is used for GLA) with the timeout of 900 seconds; (ii) perform model checking using PDR [19] (ABC command `pdr`) with the timeout of 900 seconds; (iii) if the property is proved, terminate and report k, otherwise increase k by 1, and peform steps $(i) - (iii)$ again. If an algorithm cannot produce an abstraction that can be proven within the allowed iterations (k from 1 to 99), abstraction depth is marked as >99.

C. Discussion of the initial results

The results of the initial set of experiments are presented in the left and the centre plots of Figure 1 and in Table I. The left plot in Figure 1 presents the comparison of the sizes of abstractions computed by GLA and GLAm — for each instance the two algorithms were run to the same frame number, and the size of abstraction, divided by the initial size of the instance, is plotted. The plot demonstrates that on the vast majority of the testcases the abstractions constructed by GLAm are smaller. However, what is unexpected, is that for some testcases the resulting abstraction is actually *larger* with minimization than without it. This motivated a deeper study of the impact of the minimization on the abstraction size — we discuss the results in Section VII.

It should come as no surprise that MUS-based core minimization does not come for free. The centre plot of Figure 1 compares the run-times of the GLA algorithm with and without core minimization. We observe that although for the majority of the instances the overhead is within the factor of 2, in some cases the overhead can be very significant.

So, is it worth to use MUS-based minimization in GLA ? To answer this question, we looked at the impact of minimization on the total verification time, that is, the time for abstraction plus the time for the subsequent proof (with PDR), if one could be constructed. The results of the comparison are presented in Table I. In the table, columns **#Obj**, **Dep**, **AbsT**, **PdrT** and **TotT** denote the number of AIG Objects, BMC depth, CPU time for abstraction, CPU time for proving using PDR and total CPU Time required for verification, respectively. We observe that for the easier testcases minimization leads to an increase in the verification time. However, for the difficult instances the situation is very different. Consider, for example, `6s8` — the precise abstraction was constructed in half the time, and one third of the frame bound, and, more significantly, the proof engine handled the abstraction from GLAm in just under 500 seconds, while the abstraction produced without minimization could only be proven in over 1400 seconds, giving, in total, 3x reduction in verification time. On `intel011` and `intel015`, the gain from minimization was 2x. Finally, GLA was unable to construct a precise abstraction for `intel031` in under 100 frames, while GLAm has built and proved the abstraction in 500 seconds. Notice that on `neclaftp1002` we observe 13x speed-up in total verification time, due to reduced abtraction computation time.

The current set of results suggests that MUS-based minimization can provide a significant performance boost on difficult to verify testcases.

VII. Minimization and the Size of Abstraction

As discussed in Sec. V the effort put into the core minimization procedure can be controlled in a number of ways. One of the possibilities is to adjust the time limit for each call to `Minimize` in Alg. 1, in effect performing less precise minimization at higher time-frames. This effect can be replicated by controlling the start and end time-frames for minimization instead. By GLAm$[\alpha, \beta]$ we refer to the version of GLAm algorithm where the minimization is performed only

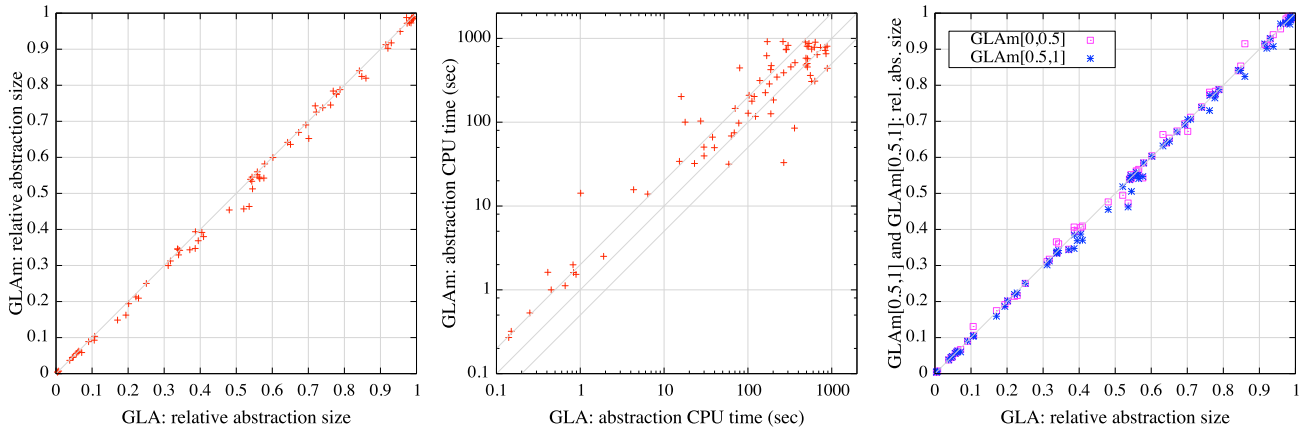

Fig. 1. Comparison between various abstraction algorithms: left — GLA vs GLAm in terms of abstraction size (normalized to the size of the original instance); center — GLA vs GLAm in terms of CPU time; right — GLA vs GLAm$[0, 0.5]$ and GLA vs GLAm$[0.5, 1]$ in terms of abstraction size (normalized).

TABLE I
DETAILED COMPARISON BETWEEN GLA AND GLAM

HWMCC	Original	GLA					GLAm				
Name	#Obj	Dep	#Obj	AbsT	PdrT	TotT	Dep	#Obj	AbsT	PdrT	TotT
6s19	14916	8	943	0.82	2.40	**3.22**	8	932	2.03	3.03	5.06
6s8	3413	92	2393	638.21	831.75	1469.96	39	2259	359.40	140.17	**499.57**
6s9	16163	8	976	0.82	3.33	**4.15**	8	963	1.60	3.79	5.39
intel011	8767	58	3385	85.15	474.24	559.39	64	3065	153.94	100.27	**254.21**
intel015	8293	64	3355	66.97	685.68	752.65	66	3249	136.20	356.01	**492.21**
intel018	6363	58	2510	40.79	57.56	98.35	58	2346	63.90	33.31	**97.21**
intel019	6611	60	2709	55.28	48.36	103.64	58	2513	84.09	17.42	**101.51**
intel020	5603	52	1888	21.52	12.16	**33.68**	54	1952	38.42	17.46	55.88
intel021	5739	54	1941	22.06	14.75	**36.81**	56	1993	42.43	24.26	66.69
intel022	8573	62	2914	98.45	15.65	**114.10**	62	2824	160.76	28.59	189.35
intel023	5585	60	2073	45.48	11.94	**57.42**	60	1919	61.26	8.99	70.25
intel024	5570	58	1911	35.45	5.96	**41.41**	58	1905	55.37	19.60	74.97
intel026	5955	48	1890	14.95	2.08	**17.03**	48	1861	26.35	1.79	28.14
intel029	8610	58	2679	91.04	12.36	**103.40**	58	2581	131.57	7.81	139.38
intel031	8627	>99	-	-	-	-	85	4422	346.94	186.51	**533.45**
intel062	9895	23	7321	17.35	191.32	**208.67**	23	7298	39.58	184.57	224.15
neclaftp1001	71264	7	12205	2.91	0.31	**3.22**	6	10591	9.86	0.22	10.08
neclaftp1002	71264	17	14154	296.94	4.46	301.40	9	11571	22.10	0.17	**22.27**
neclaftp2001	43779	22	10027	233.79	7.44	241.23	15	9181	211.49	2.24	**213.73**
neclaftp2002	43779	19	9702	335.67	4.38	340.05	20	9320	243.74	6.59	**250.33**

TABLE II
DETAILED COMPARISON BETWEEN THE VARIOUS ALGORITHMS

HWMCC	GLA	GLAm					GLAm$[0, 0.5]$					GLAm$[0.5, 1]$				
Name	TotT	Dep	#Obj	AbsT	PdrT	TotT	Dep	#Obj	AbsT	PdrT	TotT	Dep	#Obj	AbsT	PdrT	TotT
6s19	**3.22**	8	932	2.03	3.03	5.06	8	937	1.01	2.96	3.97	8	939	3.22	2.89	6.11
6s8	1469.96	39	2259	359.40	140.17	499.57	39	2321	183.91	101.64	285.55	39	2388	160.15	113.24	**273.39**
6s9	**4.15**	8	963	1.60	3.79	5.39	8	965	0.88	3.47	4.35	8	969	1.85	3.49	5.34
intel011	559.39	64	3065	153.94	100.27	**254.21**	62	3385	113.81	427.90	541.71	64	3062	150.60	136.41	287.01
intel015	752.65	66	3249	136.20	356.01	492.21	>99	-	-	-	-	66	3211	181.86	138.00	**319.86**
intel018	98.35	58	2346	63.90	33.31	97.21	58	2510	51.65	63.41	115.06	58	2347	53.65	19.64	**73.29**
intel019	103.64	58	2513	84.09	17.42	**101.51**	60	2709	74.53	55.86	130.39	60	2464	79.92	43.31	123.23
intel020	**33.68**	54	1952	38.42	17.46	55.88	54	2103	30.45	12.93	43.38	52	1873	39.16	17.97	57.13
intel021	36.81	56	1993	42.43	24.26	66.69	54	1926	18.00	18.35	**36.35**	54	1917	22.95	15.42	38.37
intel022	114.10	62	2824	160.76	28.59	189.35	60	2879	63.59	16.05	**79.64**	62	2910	97.95	21.49	119.44
intel023	57.42	60	1919	61.26	8.99	70.25	60	1927	45.05	11.04	**56.09**	60	1919	57.91	9.63	67.54
intel024	**41.41**	58	1905	55.37	19.60	74.97	58	2006	53.47	15.58	69.05	58	1902	60.99	11.01	72.00
intel026	17.03	48	1861	26.35	1.79	28.14	48	1890	13.72	2.02	**15.74**	48	1861	16.40	1.64	18.04
intel029	103.40	58	2581	131.57	7.81	139.38	58	2679	102.66	13.00	115.66	58	2597	81.38	10.19	**91.57**
intel031	-	85	4422	346.94	186.51	533.45	>99	-	-	-	-	58	4362	178.99	47.40	**226.39**
intel062	208.67	23	7298	39.58	184.57	224.15	23	7308	26.57	122.22	**148.79**	23	7306	31.37	152.03	183.40
neclaftp1001	**3.22**	6	10591	9.86	0.22	10.08	7	10592	15.45	0.32	15.77	7	12068	6.48	0.30	6.78
neclaftp1002	301.40	9	11571	22.10	0.17	**22.27**	12	12037	22.74	0.53	23.27	20	14150	350.96	2.30	353.26
neclaftp2001	241.23	15	9181	211.49	2.24	213.73	15	9482	150.34	4.43	**154.77**	22	10026	322.05	7.23	329.28
neclaftp2002	340.05	20	9320	243.74	6.59	250.33	22	9350	158.27	4.61	**162.88**	19	9670	298.88	17.93	316.81

on frame numbers f, such that $\alpha k \leq f \leq \beta k$, where k is the number of frames passed to GLAm. The scatter plot on the right of Figure 1 compares the sizes of the abstractions produced by GLAm$[0, 0.5]$ (whereby minimization is performed only in the first half of the frames) and GLAm$[0.5, 1]$ (minimization is only during the second half of the frames)

with the size of abstraction produced by GLA. Interestingly, while the minimization on the second half produces smaller abstraction in all of the cases, the minimization on the first half of the frames can sometimes produce larger abstractions than without the minimization at all. We conjecture that this effect is due to the fact that the aggressive minimization at the beginning "squeezes" abstraction too much, forcing the abstraction algorithm to compensate for the missing logic once the minimization is disabled. Since, at that point, cores are not minimized anymore, the cores from the SAT solver appear to become less precise.

Nevertheless, as demonstrated in Table II the strategy seems to work well for the easier testcases. However, the best results for the difficult test cases are obtained with GLAm$[0.5, 1]$ which minimizes the cores in the second half of the frame-range. Whereas GLA vs. GLAm results in 11 vs. 9 wins, the use of GLAm$[\alpha, \beta]$ yields further improvements over GLA, namely, 5 vs. 3, 7 and 5 wins for the different versions of GLAm$[\alpha, \beta]$. To gain a better insight into the effects of varying the minimization effort on the size of abstraction we ran an extended set of experiments on some of the more difficult instances. In these experiments we ran GLAm$[0, \alpha)$ for the values of $\alpha = 0, 0.1, \ldots, 1$, with $[0, 0)$ denoting no minimization, and $[0, 1)$ denoting full minimization, as well as GLAm$(1 - \beta, 1]$ for the same values of β. Thus, the "alpha" experiments minimize starting from the beginning for larger and larger intervals, as α grows, while the "beta" experiments minimize at the end, again for larger and larger intervals, as β grows. Figure 2 presents the results for two instances: `6s8` and `intel011`. We note that as we start minimization from the beginning (red and blue lines), in both cases there is a peak in the abstraction size, and as the minimization interval grows, the abstraction gets tightened, although not proportionally to the size of the interval. Eventually, the abstraction reaches small size, although, for `6s8`, not the smallest possible. The impact of minimization at the end of the interval is entirely different (green and purple lines) — initially, the size of abstraction does not change, but at a certain point ($\beta = 0.1$ for `intel011`, $\beta = 0.6$ for `6s8`) there is a sharp drop in the abstraction size, however for `6s8` some of the reductions disappear as we approach the full range.

Clearly, this behaviour requires a deeper analysis. However, at the moment, it appears that minimization on the second half of the interval is somewhat more robust — this is also supported by the results in Table II.

VIII. Conclusions

This paper studies the use of core minimization using MUS in SAT-based abstraction algorithms. Experimental results on difficult industrial testcases from HWMCC'11 show that MUS-based core minimization can produce significant reductions in proof time.

An interesting future research topic is to apply MUS-based minimization *during* the refinement (i.e. in the inner while-loop of Alg. 1). Although currently GLA is using priority-based refinement, it can be easily modified to perform SAT based refinement. In this case, whenever each of the counterexamples is refuted, MUS-based minimization can be used immediately to clean the abstraction, before it gets polluted with a large number of useless logic.

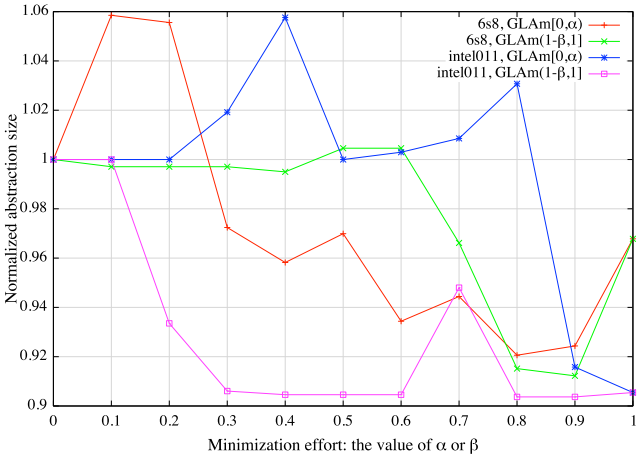

Fig. 2. Relationship between the minimization effort and the size of the resulting abstraction.

Acknowledgements We thank the anonymous referees for helpful comments. This work is partially supported by SFI grant BEACON (09/IN.1/I2618), and by FCT grants AT-TEST (CMU-PT/ELE/0009/2009) and POLARIS (PTDC/EIA-CCO/123051/2010).

References

[1] A. Gupta, M. Ganai, Z. Yang, and P. Ashar, "Iterative abstraction using SAT-based BMC with proof analysis," in *ICCAD*, 2003, pp. 416–423.
[2] K. L. McMillan and N. Amla, "Automatic Abstraction without Counterexamples," in *TACAS*, 2003, pp. 2–17.
[3] J. Marques-Silva, "Minimal unsatisfiability: Models, algorithms and applications," in *ISMVL*, 2010, pp. 9–14.
[4] A. Belov, I. Lynce, and J. Marques-Silva, "Towards efficient MUS extraction," *AI Communications*, vol. 25, no. 2, pp. 97–116, 2012.
[5] V. Ryvchin and O. Strichman, "Faster Extraction of High-Level Minimal Unsatisfiable Cores," in *SAT*, 2011, pp. 174–187.
[6] C. H. Papadimitriou and D. Wolfe, "The complexity of facets resolved," *J. Comput. Syst. Sci.*, vol. 37, no. 1, pp. 2–13, 1988.
[7] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla, "GLA: Gate-Level Abstraction Revisited," in *DATE*, 2013.
[8] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *CAV*, 2010, pp. 24–40.
[9] N. Een, A. Mishchenko, and N. Amla, "A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction Algorithm," in *FMCAD*, 2010, pp. 181–188.
[10] A. Biere, "Hardware model checking competition 2011," http://fmv.jku.at/hwmcc11, 2011.
[11] A. Nadel, "Boosting minimal unsatisfiable core extraction," in *FMCAD*, 2010, pp. 221–229.
[12] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, 1999, pp. 193–207.
[13] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla, "Variable Time-Frame Abstraction," in *IWLS*, 2012.
[14] D. Wang, P. Jiang, J. Kukula, Y. Zhu, T. Ma, and R. Damiano, "Formal property verification by abstraction refinement with formal, simulation and hybrid engines," in *DAC*, 2001, pp. 35–40.
[15] N. Amla and K. McMillan, "A hybrid of counterexample-based and proof-based abstraction," in *FMCAD*, 2004, pp. 260–274.
[16] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *J. Autom. Reasoning*, vol. 40, 2008.
[17] J. Marques-Silva and I. Lynce, "On improving MUS extraction algorithms," in *SAT*, 2011, pp. 159–173.
[18] A. Belov and J. Marques-Silva, "MUSer2: An Efficient MUS Extractor," *JSAT*, 2012, in press.
[19] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.
[20] A. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.