Recent Improvements in the SMT Solver iSAT*

Karsten Scheibler
Albert Ludwigs Universität
Freiburg
scheibler@informatik.uni-freiburg.de

Stefan Kupferschmid
Albert Ludwigs Universität
Freiburg
skupfers@informatik.uni-freiburg.de

Bernd Becker
Albert Ludwigs Universität
Freiburg
becker@informatik.uni-freiburg.de

Abstract

Over the past decades embedded systems have become more and more complex. Furthermore, besides digital components they now often contain additional analog parts – making them to embedded hybrid systems. If such systems are used in safety critical environments, a formal correctness proof is highly desirable. The SMT solver iSAT aims at solving boolean combinations of linear and non-linear constraint formulas (including transcendental functions), and thus is suitable to verify safety properties of systems consisting of both, linear and non-linear behaviour. To keep up with the ever increasing complexity of these systems we present recent improvements in various parts of iSAT. Experiments demonstrate that iSAT with all the presented improvements integrated typically gains a speedup between one and two orders of magnitude.

1. Introduction

The complexity of embedded systems increased dramatically over the past decades. The usage of these systems in safety critical environments calls for more and more sophisticated analysis techniques. An increasing number of applications in particular in the verification area are applying so-called *SAT Modulo Theories* solvers (SMT solvers) for finding bugs in systems that e.g. can be described by boolean combinations of linear constraints. However, today's embedded systems are a hybrid of digital components and analog parts. When modeling such hybrid systems,

*This work has been partially supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, http://www.avacs.org/

boolean combinations of non-linear arithmetic functions (including transcendental functions like $\sin$, $\cos$, and $\exp$) arise naturally. That is one main reason the iSAT algorithm has been invented in [FHT+07, Her11]. The SMT solver iSAT can be used for determining the satisfiability of formulas containing arbitrary boolean combinations of linear and non-linear constraints. With its *bounded model checking* layer iSAT is able to verify invariant safety properties for systems containing non-linear dynamics. As the performance of BMC procedures depends heavily on the underlying solver it is a very important task to speedup these complex pieces of software permanently. In this paper we will present the latest improvements done to iSAT. These optimizations range from preprocessing a given input formula to deep integrated solver core routines.

The paper is structured as follows. After introducing iSAT in Section 2, we will describe the enhancements applied to the formula preprocessing and to the solver core of iSAT in Section 3 and 4. In Section 5 we discuss the experimental results before we conclude with a summary.

2. The SMT Solver iSAT

This section provides a short description of iSAT. It is a complex tool and describing it in detail goes far beyond the scope of this paper. However, a basic understanding of iSAT is needed before we present our optimizations. For more information on iSAT refer to [FHT+07, Her11].

iSAT is designed for solving complex formulas containing rich arithmetic constraints. It extends the classical *conflict-driven clause learning* (CDCL) [SS96] framework by *interval constraint propagation* (ICP, cf. [BG06] for an extensive survey). ICP enables iSAT to reason about the linear and non-linear constraints with the help of interval arithmetic. In order to provide an easy to use interface iSAT allows arbitrary boolean combinations of linear and non-linear constraints as input formulas. As the underlying solver core operates on formulas in *conjunctive normal form* (CNF) the front-end has to rewrite the input formula by applying a so-called Tseitin-transformation [Tse68]. In order to enable the use of ICP every arithmetic constraint has to be rewritten by introducing fresh auxiliary variables for the values of unary and binary arithmetic subexpressions. Finally the front-end will send a CNF of the following syntax to the CDCL solver core of iSAT:

$$
\begin{array}{rcl}
\mathit{formula} & ::= & \{\mathit{clause} \wedge\}^* \mathit{clause} \\
\mathit{clause} & ::= & (\{\mathit{simple\_bound} \vee\}^* \mathit{simple\_bound}) \mid (\mathit{primitive\_constraint}) \\
\mathit{simple\_bound} & ::= & \mathit{variable} \sim \mathit{rational} \mid \mathit{boolean\_literal} \\
\sim & ::= & < \mid \leq \mid \geq \mid > \\
\mathit{primitive\_constraint} & ::= & \mathit{variable} = \mathit{uop}\ \mathit{variable} \mid \mathit{variable} = \mathit{variable}\ \mathit{bop}\ \mathit{variable} \mid \\
 & & \min(\mathit{variable}, \mathit{variable}) \mid \max(\mathit{variable}, \mathit{variable}) \\
\mathit{bop} & ::= & + \mid - \mid * \\
\mathit{uop} & ::= & - \mid \mathrm{abs} \mid \sin \mid \cos \mid \exp \mid \sqrt[n]{\cdot} \mid \cdot^n \qquad (n \text{ is fixed and } n \in \mathbb{N})
\end{array}
$$

A $\mathit{variable}$ is either an integer-valued or a real-valued variable, further a $\mathit{rational}$ ranges over the rational constants, while uop and bop are the unary and binary operators. To provide a better picture of how the rewriting process operates we will show how the formula $((\sin(y) + \sqrt{x} \geq y) \Rightarrow b)$ is transformed by the front-end. First, every linear and non-linear constraint is converted in such a way that the right-hand side (RHS) of each equation is a constant. This is done by subtracting the RHS of each equation from the left-hand side (LHS), so the constraint $(\sin(y) + \sqrt{x} \geq y)$ is rewritten to $(\sin(y) + \sqrt{x} - y \geq 0)$. Second, fresh auxiliary variables for the unary and binary arithmetic

expressions are added. In this case the front-end introduces four auxiliary real-valued variables h_1 to h_4 and four unit-clauses each containing a so-called *primitive_constraint* to represent the LHS of the inequality. Together with the implication we get the following conjunction:

$$(h_4 \geq 0 \Rightarrow b) \wedge (h_1 = \sin(y)) \wedge (h_2 = \sqrt{x}) \wedge (h_3 = h_1 + h_2) \wedge (h_4 = h_3 - y)$$

This nearly matches the introduced syntax – except the first conjunct $(h_4 \geq 0 \Rightarrow b)$. The boolean structure of any given input formula is converted by the above mentioned Tseitin-transformation. That is why a fresh boolean variable h_5 is added for encoding the implication. The clauses are computed by transforming the equivalence $h_5 \Leftrightarrow (h_4 \geq 0 \Rightarrow b)$ into the following clauses $\left(\overline{h_5} \vee h_4 < 0 \vee b\right) \wedge (h_5 \vee h_4 \geq 0) \wedge \left(h_5 \vee \overline{b}\right)$. Third, the front-end has to append (h_5) to the generated clauses. The final CNF passed to iSAT's solve routine looks like:

$$\begin{aligned}&(h_5) \wedge \left(\overline{h_5} \vee h_4 < 0 \vee b\right) \wedge (h_5 \vee h_4 \geq 0) \wedge \left(h_5 \vee \overline{b}\right) \wedge \\ &(h_1 = \sin(y)) \wedge (h_2 = \sqrt{x}) \wedge (h_3 = h_1 + h_2) \wedge (h_4 = h_3 - y)\end{aligned}$$

The definition of satisfaction is standard, e.g. a CNF is satisfied by an assignment if every clause is satisfied – a clause is satisfied if at least one literal is satisfied. An interval valuation $\rho : \mathit{Var} \to \mathbb{I}_{\mathbb{R}}$ is a mapping from a set of variables *Var* to a set of intervals $\mathbb{I}_{\mathbb{R}}$. A unit clause containing a primitive constraint c is *inconsistent* under an interval valuation ρ, iff no values in the intervals $\rho(x)$ of the variables x in c can satisfy c, i.e.

$$\begin{array}{lcll}\neg\exists v \in \rho(x) & : & v \sim r \text{ if } c = (x \sim r), & (r \in \mathbb{Q}) \\ \neg\exists v \in \rho(x), \neg\exists v' \in \rho(\circ y) & : & v \sim v' \text{ if } c = (x = \circ y), & (\circ \in \{-, \text{abs}, \sin, \cos, \exp, \sqrt[n]{\cdot}, \cdot^n\}) \\ \neg\exists v \in \rho(x), \neg\exists v' \in \rho(y \circ z) & : & v \sim v' \text{ if } c = (x = y \circ z) & (\circ \in \{+, -, *\})\end{array}$$

where $\sim \in \{<, \leq, \geq, >\}$. Otherwise c is *consistent* under ρ.

The solving process is comparable to a state-of-the-art CDCL-based SAT solver. Instead of manipulating a boolean assignment, like a CDCL solver does, iSAT extends this assignment to intervals. Besides decisions on boolean variables iSAT can also decide real- or integer-valued variables. This is done by splitting the current interval at its mid-point. Comparable to the CDCL algorithm decisions are propagated. In order to propagate interval splits every arithmetic expression, like $\sin$, $+$, $*$ etc. is lifted to intervals. For instance the *interval hull* of $x + y$ denoted by $\rho(x) + \rho(y)$ is defined by the smallest enclosing interval which is representable by machine arithmetic. In other words the *boolean constraint propagation* (BCP) is extended by ICP steps. In order to illustrate how ICP can be used to derive new interval borders, have a look at Figure 1. It shows the ICP step that will be performed for the primitive constraint $x = y^2$. Suppose that the current intervals are $x \in [3, 7]$, $y \in [-2, 25]$. The left-hand side of Figure 1 takes the current interval of variable x as input and computes that variable y is in $[9, 49]$. That means ICP was able to derive a stronger lower bound for the interval of y. The right-hand side illustrates how new interval bounds can be derived for variable x by starting from the current interval of variable y. As the maximum value of y is currently 25 we derive 5 as a new stronger upper bound of the interval of x.

As mentioned in Section 1 iSAT is able to verify safety properties by applying bounded model checking (BMC) [CBRZ01]. BMC originates in the verification of sequential circuits. A BMC problem consists of a predicate $INIT_0$ describing the initial states, a predicate $TRANS_{i,i+1}$ defining how variables change from step i to $i + 1$, and lastly a predicate describing the unsafe system

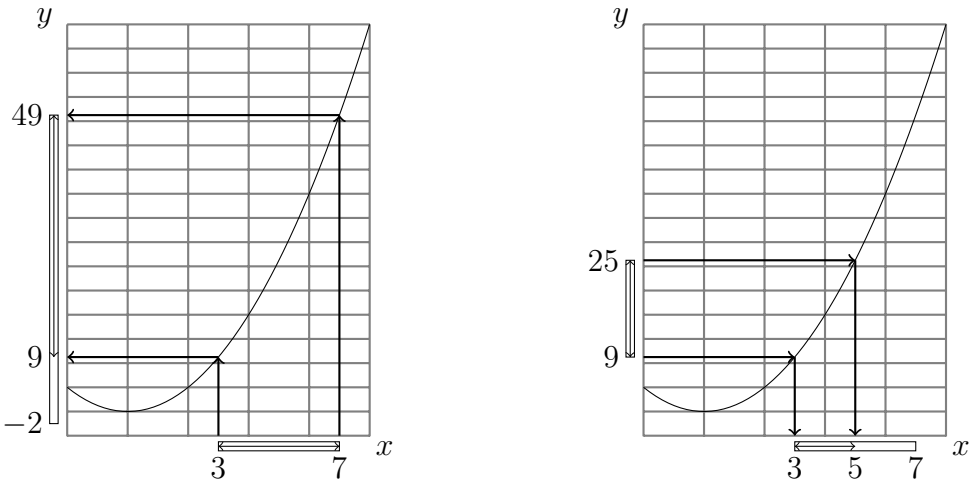

Figure 1: ICP for the primitive constraint $x = y^2$.

states $TARGET_k$. A system trace of depth k is then defined as follows:

$$\Phi_k = INIT_0 \wedge \bigwedge_{i=0}^{k-1} TRANS_{i,i+1} \wedge TARGET_k$$

The BMC approach tries to detect whether or not a system can reach an unsafe state at a certain depth. This is done by iteratively checking whether $\Phi_0, \Phi_1, \ldots \Phi_k$ are satisfiable or not. In case of sequential circuits the three predicates are described by propositional logic. Regarding iSAT the three predicates may contain boolean combinations of rich arithmetic constraints.

3. Improvements in Formula Preprocessing

Figure 2 shows an example input file, which is understood by iSAT. So far iSAT used an *abstract syntax tree* (AST) to build a representation of the input formula. The AST allowed inner nodes to have one or two children. Because of its tree structure each inner node had exactly one parent. That made it difficult to find and simplify common subexpressions. Therefore the AST is replaced by an *abstract syntax graph* (ASG). Now, inner nodes may have more than two children if the operation represented by that node is commutative and associative. Furthermore, a node may have multiple parents pointing to it. Below are the normalization and simplification rules applied to the ASG. For simplicity variable names and some small numbers are used in the following examples. Note that v, w, x, y, z are just placeholders for ASG-nodes representing arbitrary arithmetic expressions and a, b are placeholders for arbitrary boolean expressions, respectively. Every number in the ASG is represented as rational number using GMP [Gt12]. While arithmetic operations such as $\frac{1}{6} + \frac{1}{6} + \frac{1}{6}$ cause rounding errors when floating point arithmetic is used, the rational number representation in the ASG allows exact arithmetic.

1. Normalizing equations and inequalities
   Equations and inequalities are rewritten to have a numeric constant on the right-hand side. Additionally, the left-hand side is normalized, such that the first variable has a constant factor of 1. For example $(2 \cdot x + 4 \cdot y < 10 \cdot z + 2)$ will be rewritten to $(x + 2 \cdot y - 5 \cdot z < 1)$.

```
DECL
float [-10,10] x, y;

EXPR
(abs(x^2)*x + 2*y + 2*x < 7) and (0.5*x*x*x + x + y <= 3);
```

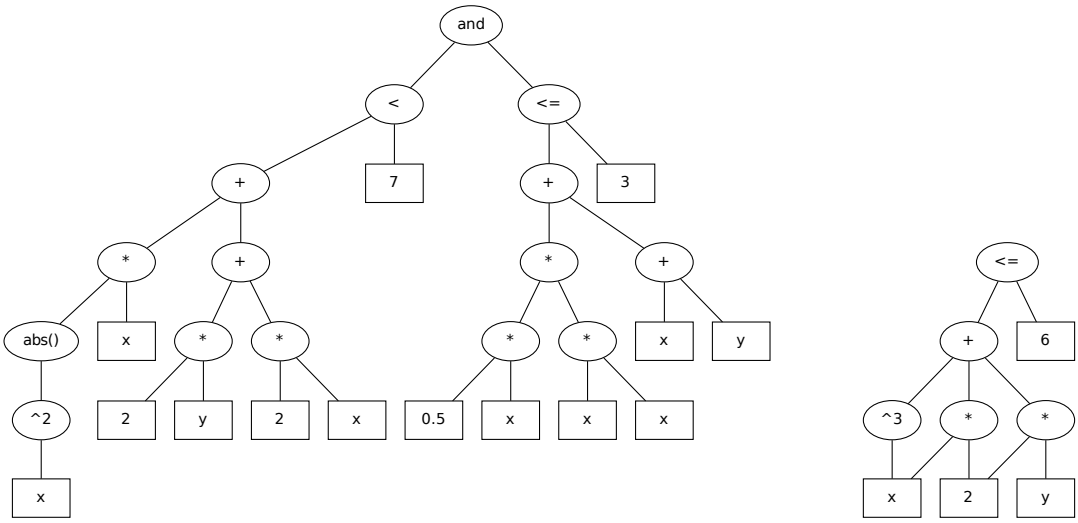

Figure 2: An example input file for iSAT together with its AST (left) and ASG (right) representations

2. Removing unneeded parenthesis
   For example $((x+y)+(v+w))$ is transformed into $(x+y+v+w)$. Regarding ASG-nodes this means adjacent nodes representing the same operation are joined into one node with multiple children – as long as the represented operation is commutative and associative.

3. Sorting children of a node
   If a node operation is invariant to the order of its children, the children are sorted. For example $(z+x+y)$ is rewritten to $(x+y+z)$. Having a fixed order eases the detection of isomorph nodes. Sorting is applied for nodes representing the following operations: $\wedge, \vee, \oplus, +, *$.

4. Sharing nodes
   Everytime a new node is going to be created, it is checked if there already exists such a node within the ASG containing the same operation and children. If there is already such a node this node is used instead of creating a new one.

5. Removing redundant nodes
   Here we exploit idempotence within AND- and OR-nodes. For example $(a \wedge b \wedge a)$ is simplified to $(a \wedge b)$. Furthermore, inequalities are also considered, because one inequality could subsume a second one. Here we exploit the fact that every inequality is already normalized and has a numeric constant on its right-hand side.

original expression	simplified expression
$(a \wedge a \wedge \ldots)$	$(a \wedge \ldots)$
$(a \vee a \vee \ldots)$	$(a \vee \ldots)$
$((x < 5) \wedge (x < 7) \wedge \ldots)$	$((x < 5) \wedge \ldots)$
$((x < 5) \vee (x < 7) \vee \ldots)$	$((x < 7) \vee \ldots)$
$(a \oplus a \oplus b \ldots)$	$(b \oplus \ldots)$

6. Detecting contradictions and tautologies
   If for example an AND-node contains $(a \wedge \neg a \wedge \ldots)$ it can never become `true`, because $(a \wedge \neg a)$ is always `false`. Inequalities are also considered here.

original expression	simplified expression
$(a \wedge \neg a \wedge \ldots)$	`false`
$(a \vee \neg a \vee \ldots)$	`true`
$((x < 5) \wedge (x > 7) \wedge \ldots)$	`false`
$((x < 7) \vee (x > 5) \vee \ldots)$	`true`

7. Basic arithmetic simplifications
   Furthermore, basic arithmetic rewritings are applied. Here some examples:

original expression	simplified expression
$(-(-x))$	(x)
$(\lvert -x \rvert)$	$(\lvert x \rvert)$
$(x + 2 \cdot x + 3 \cdot x)$	$(6 \cdot x)$
$(x \cdot x^2 \cdot x)$	(x^4)
$(\sqrt{x^2})$	$(\lvert x \rvert)$
$(\lvert x^2 \rvert)$	(x^2)
$\ldots$	$\ldots$

As can be seen in Figure 2, due to its enhanced simplification capabilities the ASG representation of the input formula is much smaller compared to the AST representation. As the ASG is used by iSAT's front-end to compute the primitive constraints for every arithmetic expression as explained in Section 2 a compact representation results in fewer primitive constraints and clauses. This usually results in better solving times.

Furthermore, the simplifications done in the ASG may strengthen the ICP. Especially simplifications like $(x \cdot x) \rightsquigarrow (x^2)$ may result in tighter bounds during ICP. For example with $x \in [-10, 10]$ ICP for $(h = x^2)$ derives $h \in [0, 100]$, while for $(h = x \cdot x)$ ICP returns the weaker interval $h \in [-100, 100]$, because the routine for multiplication does not check if both operators are equal.

4. Improvements in the Solver Core

The solver core used so far is similar to the core of a SAT solver. It makes use of the two-watched-literal scheme [MMZ+01], 1UIP conflict clause creation [ZMMM01], restarts [GSCK00] and a decision heuristics similar to the Variable State Independent Decaying Sum (VSIDS) for all variable types. But some things are done differently. To unify the handling for all variable types

(boolean, integer-valued and real-valued variables), boolean variables are treated as integer variables within the bounds $[0, 1]$. So the solver core could always operate on intervals regardless of the variable type.

But being able to operate on literals directly makes it possible to apply techniques from propositional SAT solving with minor or even no modifications – for example conflict clause minimization [SB09]. On the other hand it is needed to map every interval bound to a literal. Positive literals represent bounds containing $<$ or $\leq$ as relational operator, negative literals represent bounds containing $\geq$ or $>$. To distinguish between strict and non-strict relations, every literal has an additional flag. Lets illustrate this with some small examples:

literal	flag	representing
a	*strict*	$x < 5$
$\neg a$	-	$\neg(x < 5) \rightsquigarrow x \geq 5$

literal	flag	representing
b	-	$x \leq 6$
$\neg b$	*strict*	$\neg(x \leq 6) \rightsquigarrow x > 6$

As a consequence every newly deduced interval bound requires the allocation of a new literal on-the-fly to represent that bound. At first glance this might look like a big overhead, but in fact this can be done quite efficiently. This also means that a problem only consisting of interval bounds is automatically translated to a boolean problem – ICP is not used in such a case.

Now equipped with the capability to directly operate on literals it is also very easy to support assumptions as in propositional SAT solvers [ES03]. To solve BMC benchmarks the solver core used clause groups so far (like in [MMZ+01]) to tag conflict clauses which have to be removed before solving the next BMC unrolling. Compared to clause groups assumptions are more flexible. The new solver core uses assumptions to enable and disable the unsafe states at a certain timeframe. In the following we enumerate the most prominent changes in the solver core:

1. Adapted decision heuristics
   The heuristics used in the new solver always decides existing literals before doing an interval split on an integer- or real-valued variable. The previous solver core always did a split when a decision was needed. Lets explain this with a small example in more detail. Assume the input formula is $((x + y >= 5) \vee \ldots)$ with $x, y \in [-10, 10]$. The new solver core would get $((h = x + y) \wedge (\neg a \vee \ldots))$ with a being the positive literal representing $(h < 5)$. The initial interval for the auxiliary variable h is $[-20, 20]$. When the new solver core does a decision it first assigns values to all known literals – in this case it decides if a or $\neg a$ has to be `true`, so the interval for h is either $[-20, 5)$ or $[5, 20]$. The previous solver core does not have this mapping from interval bounds to literals and would split the interval of h at its mid-point, resulting in either $[-20, 0)$ or $[0, 20]$. This change seems subtle at first glance, but it has a great impact as Section 5 shows.

2. Minimization of the conflict clause
   Beside the "normal" conflict clause minimization, the recursive variant [SB09] is also implemented and used by default.

3. Using the LBD-scheme when deleting conflict clauses
   Instead of measuring the activity of a learned clause – that means how often the clause is involved in other conflicts – the literal blocks distance (LBD) [AS09] counts once how many different decision levels are contained in a generated conflict clause. The less decision levels the more important the conflict clause.

4. Adding a blocking literal to each watch-list entry
   An entry in the watch-list usually contains a pointer to a clause. Dereferencing this pointer causes a miss in the CPU cache in most cases. Furthermore, it can be observed that many clauses are already satisfied. So if this is known beforehand the clause access could be omitted. Therefore a so-called blocking literal [CHS09] is added to the watch-list entry. This literal is an arbitrary literal taken from the clause belonging to the pointer of the watch-list entry. If this literal is `true`, dereferencing the clause pointer is not needed.

5. Save the phase of a boolean variable
   When an assignment to a boolean variable is done, the assigned phase (either positive literal or negative literal) is saved. If later on – after a backtrack – a decision has to be made and a boolean variable is chosen as decision variable, the saved phase will be assigned. Phase saving [PD07] usually performs better then assigning always `false` or a random polarity to a boolean variable.

5. Experimental Results

The results are presented in Table 1 and Table 2. Three versions of iSAT are compared: *iSAT (AST)* (baseline implementation without improvements), *iSAT (ASG)* (baseline implementation, but the AST is replaced with an ASG) and *iSAT (ASG+Core)* (contains all presented improvements). We measured the runtimes with a timeout of 900 seconds over a set of boolean, linear and non-linear benchmarks. The experiments were performed on an Intel Core i7 computer using one 3.4 GHz core and 4 GB RAM.

The benchmark set used in our experiments contains for example models of the european train control system [HEFT08], a model of an aircraft collision avoidance system [TS98], and two benchmark-families modeling a laser passing a labyrinth of either reflecting obstacles (laser1) or passing a labyrinth of a combination of absorbing, refracting and reflecting obstacles (laser2).

Most of the benchmarks are bounded model checking instances. Beside the timeout the maximal BMC depth was limited to 1000 unrollings. Table 1 and Table 2 show that the formula preprocessing done by the ASG roughly gives a speedup of an order of magnitude. An extreme example is the hong benchmark-family. Here, the ASG simplification leads to a tighter interval constraint propagation, allowing the solver to recognize the unsatisfiability of the formula with only one conflict. The new solver core further boosts the performance by an additional order of magnitude. Another extreme example is water_level_monitor. This benchmark is rather simple and it is possible to do many BMC unrollings in a short time period. The new solver core is able to do 1000 unrollings in under a second, while the previous solver core needed 100 seconds.

6. Conclusion

We presented improvements in the formula preprocessing and the solver core of the SMT solver iSAT. With the proposed ASG it is possible to find more common subexpressions and do more simplifications compared to the AST that was used before. This alone gave significant lower solving times. Additionally, changing the solver core to operate on literals directly, adapting the decision heuristics and adding further techniques known from propositional SAT solving like recursive conflict clause minimization and assumptions gave a further performance boost. Overall we have been able to accelerate the solver between one and two orders of magnitude on average.

Benchmark	BMC depth	iSAT (AST)	iSAT (ASG)	iSAT (ASG+Core)
boolean BMC benchmarks				
am2910	5	2.57s	0.70s	0.14s
b10.inv.prop	8	121.26s	26.23s	1.48s
b11.inv1.prop	65	TIMEOUT	TIMEOUT	21.86s
b12.inv1.prop	14	TIMEOUT	TIMEOUT	25.51s
b13-inv1-counter-example-depth-10	10	5.36s	1.61s	0.32s
bpb.inv.prop	9	663.17s	82.63s	2.31s
matrix.inv10.prop	10	TIMEOUT	TIMEOUT	12.39s
matrix.inv8	8	172.80s	66.95s	0.91s
vsaR.inv.prop	23	18.75s	20.70s	1.00s
linear and non-linear BMC benchmarks without transcendental functions				
laser1.01	17	TIMEOUT	11.61s	4.49s
laser1.02	19	TIMEOUT	89.63s	20.32s
laser1.03	7	TIMEOUT	0.18s	0.06s
laser1.04	19	11.15s	7.06s	2.56s
laser1.05	29	TIMEOUT	221.63s	25.93s
laser1.06	21	TIMEOUT	14.06s	2.34s
laser1.07	25	TIMEOUT	70.69s	14.49s
laser1.08	23	TIMEOUT	9.25s	2.09s
laser1.09	19	TIMEOUT	10.18s	1.96s
laser1.10	9	167.39s	3.17s	0.81s
laser1.11	31	TIMEOUT	254.29s	38.21s
laser1.12	22	353.43s	2.60s	0.50s
laser1.13	31	TIMEOUT	90.61s	18.12s
laser1.14	29	TIMEOUT	54.27s	13.21s
laser1.15	19	TIMEOUT	16.04s	4.03s
bouncing_ball_euler	57	64.02s	4.19s	1.98s
water_level_monitor	1000	106.73s	89.77s	0.46s
arbiter	5	TIMEOUT	TIMEOUT	330.65s
hysat_6_bus	4	TIMEOUT	388.71s	363.91s
minigolf	13	TIMEOUT	13.52s	3.92s
minigolf2	14	TIMEOUT	TIMEOUT	720.62s
minigolf3	9	TIMEOUT	91.27s	13.08s
osc3	50	TIMEOUT	TIMEOUT	816.36s
etcs_train_system_mod	65	MEMOUT	TIMEOUT	495.35s
train_distance_ctrl1	19	MEMOUT	2.66s	3.41s
train_system	9	14.66s	0.08s	0.06s
train_system1	33	TIMEOUT	36.53s	39.22s
train_system2	17	83.19s	5.99s	3.47s
train_system3	64	TIMEOUT	TIMEOUT	319.42s

Table 1: Comparing the runtimes (in seconds) between the different iSAT versions. The implementation used so far is named *iSAT (AST)*. The comlumn *iSAT (ASG)* shows the improvement the ASG is contributing. The last column *iSAT (ASG+Core)* contains the runtimes if the ASG and the new core are used. A timeout of 900 seconds was used.

Benchmark	BMC depth	iSAT (AST)	iSAT (ASG)	iSAT (ASG+Core)
non-linear benchmarks without transcendental functions				
hong1	-	0.01s	0.01s	0.01s
hong5	-	0.32s	0.01s	0.01s
hong10	-	117.81s	0.01s	0.01s
hong20	-	TIMEOUT	0.01s	0.01s
non-linear BMC benchmarks with transcendental functions				
laser2.01	5	0.93s	1.17s	0.33s
laser2.02	5	1.20s	0.99s	0.49s
laser2.03	15	TIMEOUT	42.89s	17.10s
laser2.04	15	560.94s	38.42s	5.15s
laser2.05	13	TIMEOUT	33.41s	6.93s
laser2.06	29	TIMEOUT	750.93s	483.75s
laser2.07	11	50.57s	2.10s	0.70s
laser2.08	13	148.56s	19.79s	1.90s
laser2.09	19	TIMEOUT	279.56s	62.52s
laser2.10	27	TIMEOUT	701.34s	55.93s
laser2.11	9	9.14s	5.10s	1.20s
laser2.12	17	TIMEOUT	96.97s	52.37s
laser2.13	17	TIMEOUT	101.19s	2.80s
laser2.14	21	156.19s	50.49s	2.29s
laser2.15	19	TIMEOUT	109.79s	3.69s
tomlin_aircraft_roundabout	3	TIMEOUT	TIMEOUT	43.97s

Table 2: Comparing the runtimes as in Table 1

References

[AS09] Audemard, Gilles and Laurent Simon: *Predicting learnt clauses quality in modern sat solver*. In *Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, pages 399–404, jul 2009.

[BG06] Benhamou, F. and L. Granvilliers: *Continuous and Interval Constraints*. In *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, pages 571–603. 2006.

[CBRZ01] Clarke, Edmund M., Armin Biere, Richard Raimi, and Yunshan Zhu: *Bounded model checking using satisfiability solving*. Formal Methods in System Design, 19(1):7–34, 2001. `citeseer.ist.psu.edu/clarke01bounded.html`.

[CHS09] Chu, Geoffrey, Aaron Harwood, and Peter J. Stuckey: *Cache conscious data structures for boolean satisfiability solvers*. JSAT, 6(1-3):99–120, 2009.

[ES03] Eén, Niklas and Niklas Sörensson: *An extensible sat-solver*. In Giunchiglia, Enrico and Armando Tacchella (editors): *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003, ISBN 3-540-20851-8.

[FHT+07] Fränzle, Martin, Christian Herde, Tino Teige, S. Ratschan, and Tobias Schubert: *Efficient solving of large non-linear arithmetic constraint systems with complex*

boolean structure. Journal on Satisfiability, Boolean Modeling, and Computation, 1(3-4):209–236, 2007.

[GSCK00] Gomes, Carla P., Bart Selman, Nuno Crato, and Henry A. Kautz: *Heavy-tailed phenomena in satisfiability and constraint satisfaction problems*. J. Autom. Reasoning, 24(1/2):67–100, 2000.

[Gt12] Granlund, Torbjörn and the GMP development team: *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. `http://gmplib.org/`.

[HEFT08] Herde, Christian, Andreas Eggers, Martin Fränzle, and Tino Teige: *Analysis of hybrid systems using hysat*. In *ICONS*, pages 196–201. IEEE Computer Society, 2008.

[Her11] Herde, Christian: *Efficient solving of large arithmetic constraint systems with complex Boolean structure: proof engines for the analysis of hybrid discrete-continuous systems*. PhD thesis, 2011, ISBN 978-3-8348-1494-4.

[MMZ+01] Moskewicz, Matthew W., Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik: *Chaff: Engineering an efficient sat solver*. In *DAC*, pages 530–535. ACM, 2001, ISBN 1-58113-297-2.

[PD07] Pipatsrisawat, Knot and Adnan Darwiche: *A lightweight component caching scheme for satisfiability solvers*. In Marques-Silva, João and Karem A. Sakallah (editors): *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007, ISBN 978-3-540-72787-3.

[SB09] Sörensson, Niklas and Armin Biere: *Minimizing learned clauses*. In Kullmann, Oliver (editor): *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009, ISBN 978-3-642-02776-5.

[SS96] Silva, João P. Marques and Karem A. Sakallah: *Grasp - a new search algorithm for satisfiability*. In *ICCAD*, pages 220–227, 1996.

[TS98] Tomlin, Claire and Shankar Sastry: *Conflict resolution for air traffic management: a study in multi-agent hybrid systems*. IEEE Transactions on Automatic Control, 43:509–521, 1998.

[Tse68] Tseitin, Grigori S.: *On the complexity of derivations in propositional calculus*. In Slisenko, A. (editor): *Studies in Constructive Mathematics and Mathematical Logics*. 1968.

[ZMMM01] Zhang, Lintao, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik: *Efficient conflict driven learning in a boolean satisfiability solver*. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press, ISBN 0-7803-7249-2. `http://dl.acm.org/citation.cfm?id=603095.603153`.