The MathSAT5 SMT Solver ⋆

Alessandro Cimatti[1], Alberto Griggio[1],
Bastiaan Joost Schaafsma[1,2], and Roberto Sebastiani[2]

[1] FBK-IRST, Trento, Italy
[2] DISI, University of Trento, Italy

Abstract. MATHSAT is a long-term project, which has been jointly carried on by FBK-IRST and University of Trento, with the aim of developing and maintaining a state-of-the-art SMT tool for formal verification (and other applications). MATHSAT5 is the latest version of the tool. It supports most of the SMT-LIB theories and their combinations, and provides many functionalities (like e.g. unsat cores, interpolation, AllSMT). MATHSAT5 improves its predecessor MATHSAT4 in many ways, also providing novel features: first, a much improved incrementality support, which is vital in SMT applications; second, a full support for the theories of arrays and floating point; third, sound SAT-style Boolean formula preprocessing for SMT formulae; finally, a framework allowing users for plugging their custom tuned SAT solvers. MATHSAT5 is freely available, and it is used in numerous internal projects, as well as by a number of industrial partners.

1 Introduction

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a (typically quantifier-free) first-order formula with respect to some decidable theory $\mathcal{T}$ (or combination of theories $\bigcup_i \mathcal{T}_i$). SMT solvers have proved to be powerful and expressive backend engines for formal verification in many contexts, including the verification of software, hardware, and of timed and hybrid systems. An amount of papers with novel and very efficient techniques for SMT has been published in the last decade, and some very efficient SMT tools are now available.

MATHSAT is a long-term project, which has been jointly carried on by FBK-IRST and University of Trento in the last decade, with the aim of developing and maintaining a state-of-the-art SMT tool for formal verification (and other applications). In this paper we present MATHSAT5, the latest version of the tool. MATHSAT5 supports most of the SMT-LIB theories and their combinations, and provides many SMT functionalities (e.g. unsatisfiable cores, interpolation, AllSMT). It does not offer support for quantifiers. MATHSAT5 improves its predecessor MATHSAT4 [6] in many ways, also providing novel features. First, it provides a much improved support for *incremental solving*, which is vital in many applications of SMT (e.g., symbolic simulation, SW model checking). Second, it fully supports also the theories of *arrays* and *IEEE floating*

⋆ A. Griggio is supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION. B. Schaafsma and R. Sebastiani are supported in part by Semiconductor Research Corporation under GRC Research Project 2012-TJ-2266 WOLF.

point numbers. Third, it provides (incremental) *Theory Aware SAT-Preprocessing*, i.e. sound SAT-style Boolean formula preprocessing adapted for SMT formulas. Finally, it supplies a framework for third-party SAT-solver integration, allowing users -including industrial users- for plugging their custom tuned solvers. MATHSAT5 is available at [33], and it is used in numerous internal projects, as well as by a number of industrial partners.

The paper is structured as follows: §2 describes the functionalities of MATHSAT5; §3 discusses its architecture; §4 discusses the specifics of our implementations; §5 shows an empirical evaluation; §6 discusses a number of in-house applications of MATHSAT5; finally in §7 we draw some conclusions and discuss ongoing and future work.

2 Functional View

MATHSAT5 provides functionalities for both satisfiability checking (*solving*) and for extended SMT tasks. It can be accessed either through the command line, by feeding a `SMT-LIB` file (in either the `SMT-LIB v.1` or `SMT-LIB v.2` standard), or through an API, which is similar in spirit to the commands of the `SMT-LIB v.2` language (with additional functionalities).

Solving. MATHSAT5 solving facilities support most of the `SMT-LIB` theories of interest, including that of equality and uninterpreted functions ($\mathcal{EUF}$), that of arrays ($\mathcal{AR}$), and their combinations with the theories of linear arithmetic on the rationals ($\mathcal{LA}(\mathbb{Q})$), the integers ($\mathcal{LA}(\mathbb{Z})$) and mixed rational-integer ($\mathcal{LA}(\mathbb{QZ})$), that of fixed-width bit-vectors ($\mathcal{BV}$), and that of floating-point arithmetic ($\mathcal{FP}$). Notably, to the best of our knowledge, MATHSAT5 is one of the very few SMT solvers supporting $\mathcal{FP}$ [31].

Many SMT-based formal verification techniques (e.g., BMC, symbolic simulation, lazy abstraction) need invoking the backend SMT solver *incrementally*, in a stack-based manner, by pushing and popping sub-formulas. To cope with this fact, regardless the theories addressed, MATHSAT5 provides an *incremental*, stack-based interface, allowing multiple satisfiability checks over a changing clause database, and maintaining useful information of the status of computation (e.g. learned clause, scores) from one call to the other, which prevents restarting the search from scratch each time.

Beyond Solving. Like its predecessors, MATHSAT5 was designed primarily to be used in formal verification settings, where often simple queries for a "SAT/UNSAT" answer are not sufficient. Thus, MATHSAT5 provides several extended SMT functionalities.
Production of Models. When the input formula φ is satisfiable, MATHSAT5 can produce a satisfying interpretation $\mathcal{I}$ on domain variables, with a congruent partial interpretation of uninterpreted functions and predicates. [3]
Production of Proofs. When φ is unsatisfiable, MATHSAT5 can produce a proof, combining a resolution proof and theory-specific sub-proofs of the $\mathcal{T}$-lemmas.

[3] E.g., in $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$, if φ is $x = 5 \wedge f(x) < 3$, then $\mathcal{I}$ may assign x to 5 and $f(5)$ to 2.

Extraction of Unsatisfiable Cores. MATHSAT5 allows for extracting a $\mathcal{T}$-unsatisfiable subset of an input clause set. This implements both the standard extraction from a resolution proof, and the "lemma-lifting" approach we described in [11], which invokes an external Boolean unsat-core extractor available off-the-shelf, thus benefitting from every size-reduction techniques implemented there.
Interpolation. MATHSAT5 allows for computing (Craig) interpolants of pairs of input mutually-inconsistent SMT formulas for nearly all implemented theories. This feature includes optimized interpolant generator for $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Q})$ [10], for $\mathcal{LA}(\mathbb{Z})$ [30], for $\mathcal{BV}$ [28], and an interpolant generator for combined theories based on DTC [10].
AllSMT & Predicate Abstraction. MATHSAT5 implements an "AllSMT" functionality [32]: in case of a satisfiable input formula φ, it can efficiently enumerate a complete set of theory-consistent partial assignments satisfying φ. This feature is useful for performing predicate abstraction in a SMT-based Counter-Example-Guided Abstraction-Refinement (CEGAR) context (e.g. [3]).
Enumeration of Diverse Models. Strictly related to AllSMT, MATHSAT5 implements a brand-new functionality, which was requested from our industrial partners. The users are allowed to define a set of *diversifying predicates [resp. terms]* and MATHSAT5 enumerates models which differ to one another for the truth value [resp. domain value] of at least one of these predicates [resp. terms]. [4] This technique is useful to, e.g., guarantee coverage of all branches in a program, partitioning the value space into a grid, cover all values of some selector variables, investigate corner cases, etc.
Pluggable SAT Solvers. Finally, MATHSAT5 provides an API for integrating external SAT-solvers, allowing (industrial) users for plugging their custom tuned solver for their specific applications.

MathSAT5 vs. MathSAT4. MATHSAT5 extends and improves its predecessor MATHSAT4 in many ways.

From the perspective of SMT solving, a full support for the theories of arrays ($\mathcal{AR}$) and floating point ($\mathcal{FP}$) has been introduced; the solvers for $\mathcal{BV}$ and $\mathcal{LA}(\mathbb{Z})$ have been re-implemented and made much more efficient, and the latter has been extended to deal also with mixed rational-integers $\mathcal{LA}(\mathbb{QZ})$. The default underlying SAT solver has been improved. Moreover, (incremental) *Theory Aware SAT-Preprocessing*, i.e. sound SAT-style Boolean formula preprocessing adapted for SMT formulas, has been introduced. (See next sections.) Overall, the whole tool has been redesigned to fully support incrementality, in both solving and other functionalities.

From the perspective of SMT functionalities, *Enumeration of Diverse Models* and *Pluggable SAT Solvers* are brand new. *Interpolation* has been extended to $\mathcal{LA}(\mathbb{Z})$ [30] and $\mathcal{BV}$ [28]. Finally, the *Production of Models* and of *Production of Proofs* functionalities have been significantly improved. Importantly, the *Production of Proofs*, *Extraction of Unsat Cores*, *Interpolation*, *AllSMT*, *Enumeration of Diverse Models*, *Pluggable SAT Solvers* functionalities have been adapted to work also in incremental mode (*Production of Models* was already incremental in MATHSAT4).

[4] Notice that diversifying terms are meaningful only with terms on discrete and small bounded domains, like enumeratives, bounded integers with small ranges, small-size bit-vectors.

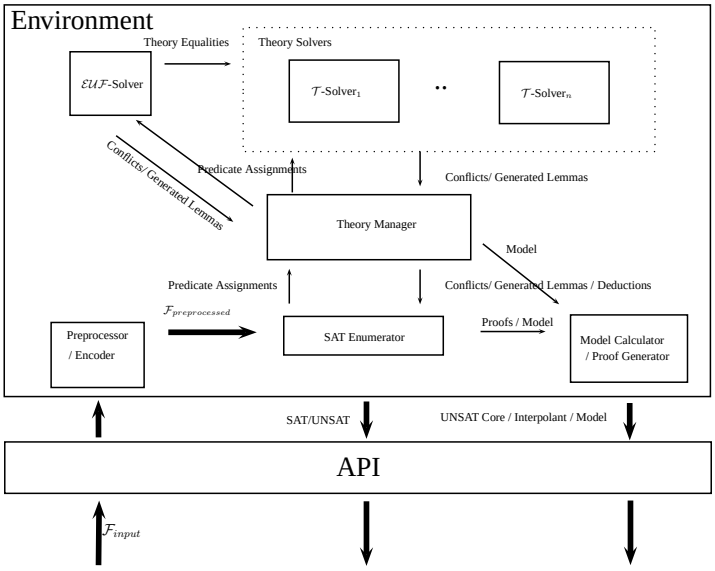

Fig. 1. Architectural overview of MATHSAT5.

3 Architectural View

Figure 1 details the MATHSAT5 architecture. From a high-level perspective, the main component of MATHSAT5 is the environment, which acts as a coordinator for the various sub-components of the solver (preprocessor, constraint encoder, theory manager, proof and model generator, SAT engine and individual theory solvers). Besides coordination of components, the environment is also responsible for various administrative tasks, such as memory management and garbage collection.

The preprocessor is a term-rewriting engine which performs formula normalization and constant inlining. In formula normalization we rewrite redundant formulas to a "simpler" or "smaller" form. This is done by applying, up to a fix point, some rewrite rules from a database. In constant inlining we replace constants with their definitions. (For example if the formula contains a predicate $(x = 3)$, we replace all occurrences of x in the input formula by 3.)

The constraint encoder performs the CNF conversion of the input formulas, as well as the encoding of various constructs which are not directly supported by the core components of MATHSAT5. For instance, it eliminates term-level if-then-else constructs $\text{ITE}(c, t, e)$ from a formula by replacing them with fresh variables x_{ITE} and by adding to the formula the clauses $(\neg c \vee (x_{\text{ITE}} = t))$ and $(c \vee (x_{\text{ITE}} = e))$.

The core of MATHSAT5 is composed of the SAT engine and the theory solvers, which interact following the standard lazy/DPLL($\mathcal{T}$) approach [2]. The SAT engine is either our native SAT engine or a "pluggable", third-party SAT engine. The former is a MINISAT-style SAT solver [21], equipped with a preprocessor/inprocessor supporting

the following Boolean formula simplifications: *Variable Elimination (VE), Subsumed clause removal (SCR)* and *Backwards subsumption (BS)* [20]. In VE we perform DP-resolution on a variable x, replacing all clauses of the form $(C \vee x)$ and $(C \vee \neg x)$ with their pairwise resolvents. In SCR, if clause C_i subsumes C_j, i.e. C_i contains a subset of the literals in C_j, then it follows the C_j can be dropped from the input formula. In BS, we take advantage from the fact that, if we resolve $(\neg x \vee C_i)$ with $(x \vee C_j)$ on x, and C_i subsumes C_j, then it follows that their resolvent equals C_j, thus we can shorten $(x \vee C_j)$ to C_j. Notice that in general (some of) these simplifications are unsound in an (incremental) SMT setting. We describe how we have adapted them to ensure correctness in §4.3.

The pluggable SAT engine allows for the integration of an external, third-party SAT solver in MATHSAT5. The architecture is based on a "SAT worker" wrapper interface for the external solver, which is required to implement a number of callback functions to respond to various events generated by the other MATHSAT5 components, and to satisfy certain requirements that are needed for a proper integration in an SMT context. For more details, we refer to §4.4.

The theory manager acts as a unified interface between the SAT engine and individual $\mathcal{T}$-solvers, allowing for a modular integration of new theories. In our architecture, individual $\mathcal{T}$-solvers know nothing about neither the SAT engine nor their sibling solvers, and they only interact with the theory manager. In this way, $\mathcal{T}$-solvers can be easily added and removed without affecting the rest of the system.

The SAT engine and the theory manager communicate with the model and proof calculator component, which is responsible of producing models for satisfiable formulas and refutation proofs for unsatisfiable ones. Refutation proofs consist of a Boolean part and a theory-specific part. The theory-specific part consists of the list of theory lemmas generated during search, together with theory-specific proofs for them. For example, for $\mathcal{LA}(\mathbb{Q})$ a proof consists of a list of inequalities and the corresponding coefficients needed for obtaining a contradiction via linear combinations, whereas for $\mathcal{EUF}$ it consists of a sequence of applications of the reflexivity, symmetry, transitivity and congruence axioms leading to the violation of some disequality. The Boolean part of the proof, computed by the SAT engine, consists instead of Boolean resolution steps among the clauses of (the CNF conversion of) the input formula and the theory lemmas generated by the $\mathcal{T}$-solvers. From the refutation proof, interpolants and/or unsatisfiable cores can then be produced (possibly with the help of an external Boolean unsat-core extractor, as described in [11]).

3.1 Solving techniques

As already mentioned above, MATHSAT5 generally uses a lazy/DPLL($\mathcal{T}$) approach for satisfiability checking: roughly speaking, the SAT engine is used as an enumerator of truth assignments that are then checked by the theory solvers for $\mathcal{T}$-consistency, until either all of them are refuted, or a $\mathcal{T}$-model is found [2]. $\mathcal{T}$-solvers are based on state-of-the-art algorithms for the various theories: congruence closure for $\mathcal{EUF}$ [35], simplex for $\mathcal{LA}(\mathbb{Q})$ [19], a layered approach based on simplex and branch and bound for $\mathcal{LA}(\mathbb{Z})$ and $\mathcal{LA}(\mathbb{QZ})$ [29], and axiom instantiation for $\mathcal{AR}$ [26]. The theory of $\mathcal{BV}$ is handled via a reduction to SAT (*bit-blasting*), performed either lazily, using a separate

SAT engine acting as $\mathcal{T}$-solver [28], or eagerly, i.e. directly using the main SAT engine. For the $\mathcal{FP}$ theory, MATHSAT5 can use either an approach based on bit-blasting (lazy or eager), or a more recent one based on a novel combination of Interval Constraint Propagation for floating-point numbers and modern Conflict-Driven Clause Learning (CDCL) SAT solvers, based on a generic framework for extending CDCL algorithms to abstract domains [31]. Finally, problems in a combination of theories are handled using our *Delayed Theory Combination (DTC)* framework [4, 7].

4 Implementation

In this Section, we provide some details on the most significant aspects of the implementation of MATHSAT5.

4.1 Low-level Optimizations

MATHSAT5 is implemented in C++, using an object-oriented paradigm. One of the most important aspects of the implementation is the use of several ad-hoc variants of common data structures (such as vectors, stacks, queues, hash tables), specialized for critical parts of the code, which significantly improve the overall performance of the solver. The main reason for this is memory management. In particular, our custom data structures and algorithms are designed to reduce the overhead due to excessive memory allocations/deallocations and to exploit the availability of specialized allocators that try to ensure a cache-friendly layout of data in memory. For a similar reason, we use our own custom written library for arbitrary-precision arithmetic, built on top of the GNU Multi Precision library [25], which avoids costly memory operations in the cases in which the numbers to manipulate fit into machine words.

One might question the value of these low-level "micro-optimizations", arguing that there are many higher-level factors (such as e.g. branching heuristics, search strategies, preprocessing algorithms) which have a much stronger impact on the performance of an SMT solver. Our experience however suggests that in practice these details have a very visible impact, in particular on scalability, which is crucial for the successful application of the solver in industrial settings. We refer to [27] for an example of the impact of low-level optimizations on the performance of MATHSAT on real-world $\mathcal{LA}(\mathbb{Q})$ formulas.

4.2 Incrementality

In an incremental setting, MATHSAT5 manipulates a *stack* $S \stackrel{\text{def}}{=} [\varphi_1, \ldots, \varphi_n]$ of formulas, which corresponds to the input problem $\varphi_1 \wedge \ldots \wedge \varphi_n$. The stack is manipulated via a *push* and *pop* interface. Pushing a formula ψ corresponds to conjoining ψ to the current input problem, whereas popping corresponds to discarding the most recently added conjunct. All the internal components of MATHSAT outlined in Figure 1 are designed to exploit this stack-based interaction. In the DPLL engine, incrementality is implemented by exploiting a variant of solving under assumptions [22]. Each element φ_i of the stack is associated to a *label literal* x_{stack_i}. During CNF conversion, all the clauses for the formula φ_i are extended with the label literal $\neg x_{\text{stack}_i}$. When the

satisfiability of the input formula is decided, DPLL is invoked with the assumptions $\{x_{\text{stack}_1}, \ldots, x_{\text{stack}_n}\}$. When a formula is popped from the stack, all clauses (including learnt clauses) that contain the last label literal $\neg x_{\text{stack}_n}$ are deleted. Importantly, all DPLL variables created after x_{stack_n} are also deleted, as well as all the corresponding internal variables in the theory solvers. This is very important in applications (such as e.g. [9]) in which hundreds of thousands of simple formulas, often totally unrelated to each other, are pushed and popped from the stack, in order to avoid cluttering the solver with irrelevant data.

4.3 Adapting SAT-level Preprocessing to incremental SMT

As stated above, MATHSAT5 supports the following SAT formula processing techniques: Variable Elimination (VE), Subsumed clause removal (SCR) and Backwards subsumption (BS). In general, these techniques are not sound when applied in an incremental SMT context. There are multiple reasons for this: 1) After model calculation, the extended SAT model which contains the values calculated for eliminated variables may be $\mathcal{T}$-inconsistent; 2) VE may eliminate label literals x_{stack_i} used for implementing incrementality; 3) Variables eliminated by VE may be reintroduced either during subsequent formula pushes, or during search; 4) Clauses which allowed us to shorten a clause through BS or eliminate a clause through SCR may no longer be implied by the input formula after a pop.

The first problem arises because, in an SMT context, variables in the SAT solver might represent theory constraints (i.e., they might be *proxies* for some $\mathcal{T}$-atom). In such cases, eliminating them has the effect of dropping some $\mathcal{T}$-constraints from the formula, which might change its satisfiability status. Our simple solution to this problem is to forbid the elimination of proxy variables (in SAT terminology, we *freeze* them).

The other three issues are not due to SMT, but rather to the use of the techniques in an incremental setting. Point 2) is problematic because label literals are necessary to correctly maintain the stack of formulas (see §4.2 above), and so they can't be eliminated from the formula. We avoid the problem by simply freezing label literals. For problem 3), we adopt a solution similar to the one described in [34]. Roughly speaking, the approach is based on saving clauses containing eliminated variables, instead of deleting them immediately, so that they can be re-added to the problem in case a previously-eliminated variable is re-added to the SAT solver. We simply remark that, unlike in the setting considered in [34], in SMT eliminated variables can be reintroduced even when incrementality is not used, because in general theory solvers are allowed to introduce new SAT variables during search (this is the case e.g. for Delayed Theory Combination [7] or for axiom instantiation [26]). Finally, regarding problem 4), we observe that freezing label literals automatically gives a solution for it. The reason is that, since we prohibit the elimination of label literals, clauses belonging to different pushes always differ in at least one literal which only occurs negatively, thus neither SCR nor BS is applicable. This solution, however, has the drawback of significantly limiting the applicability of subsumption. In fact, MATHSAT5 does something better than this, by employing the notions of contemporary and base clauses. Clause C_i is contemporary with respect to clause C_j if the highest label literal contained in C_i is created before the highest label literal contained in C_j. If C_i is contemporary to C_j, the

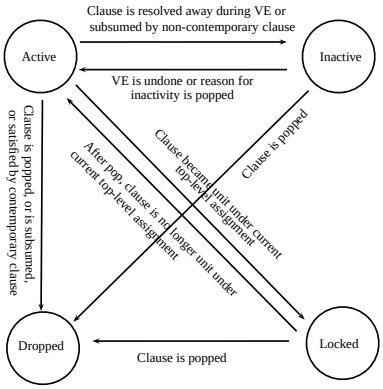

Fig. 2. Clause Management in MATHSAT5.

push/pop architecture used in MATHSAT5 ensures that as long as C_j is active, C_i is active as well. Given a clause C_i, $base(C_i)$ is the clause obtained by removing all label literals from C_i. Using these notions of contemporary and base clauses MATHSAT5 extends the SCR and BS rules as follows:

- If $base(C_i)$ subsumes $base(C_j)$ and C_i is contemporary to C_j, we can drop C_j from the input formula.
- If $base(C_i)$ subsumes $base(C_j)$ but C_i is not contemporary to C_j, we can still ignore C_j as long as C_i is active.
- If $base(C_i)$ backwards subsumes $base(C_j)$ on l and C_i is contemporary to C_j, we can shorten C_j by l.

Figure 2 summarizes the clause management system used in MATHSAT5, and shows how clauses move from being active or locked, to inactive, or dropped, depending on the circumstances.

4.4 Pluggable SAT solvers

As already described, MATHSAT5 allows for using an external CDCL-based SAT solver as its SAT engine. From the point of view of the implementation, this is achieved by 1) requiring the external solver to implement a specific *SAT worker* interface which defines the communication protocol between the external solver and the rest of MATHSAT5, and 2) requiring the external solver to invoke some *callback functions* in order to notify the rest of MATHSAT5 about specific states in the SAT search.

The SAT worker interface consists of methods for creating SAT variables, adding clauses, propagating literals deduced by the theory solvers, and retrieving the truth values of variables after a Boolean model has been found. In order to work correctly in the context of MATHSAT5, the SAT solver is required to be able to create new variables and add new clauses during search. If it uses some form of preprocessing involving variable elimination, it must also support the ability of freezing some of the variables

and correctly handle the addition of clauses containing previously-eliminated variables (see §4.3), or else preprocessing must be turned off.[5] Finally, in order to be usable in an incremental setting, the SAT solver must support solving under assumptions [22] (otherwise, it can only be used for non-incremental queries). In general, implementing such interface amounts to creating a wrapper that invokes the corresponding functions in the API of the SAT solver.[6]

Besides implementing the worker interface, the code of the external SAT solver must also be modified to invoke a number of callback functions provided by MATHSAT5, in order to allow the interaction between the SAT engine and the theory solvers in MATHSAT5 during the SAT search. In particular, the callback functions invoke the theory solvers when either a complete Boolean model or a non-conflicting partial assignment has been found. Invoking the theory solvers allows us to do early pruning, theory consistency checking and the propagation of theory deductions.

In general, the source code of the external SAT solver needs to be patched to include the proper calls to the MATHSAT5 callback functions. However, in our experience the amount of changes required is typically quite small. In our example implementations, using the MINISAT [21] and CLEANELING [18] open-source SAT solvers, the patches consist of less than 150 lines of code.

5 Experimental Evaluation

In this section, we present an experimental evaluation of MATHSAT5. We demonstrate two key properties of our solver: first, the improvement over the previous version of MATHSAT; second, the usefulness of the new features.

Benchmarks. For our experiments, we use the following classes of benchmarks.

BVuMEM. Benchmarks from the $\mathcal{BV} \cup \mathcal{AR}$ SMT-LIB category. We leave out a family containing only very large but trivial to solve benchmarks.

HSver. Benchmarks originating from practical problems in the verification of hybrid systems. The benchmarks are in the theory of $\mathcal{LA}(\mathbb{Q})$, and represent proof obligations generated by the scenario-based verification algorithms of [13]. Besides the $\mathcal{LA}(\mathbb{Q})$ component, these instances also have a complex Boolean component.

COMP09. Benchmarks from the 2009 SMT-COMP, in the categories entered by MATHSAT4 at the time.

LRA11. The application benchmarks of the 2011 SMT-COMP, for the theory of $\mathcal{LA}(\mathbb{Q})$.

The first three classes of benchmarks are considered as non-incremental (i.e. we check satisfiability once per benchmark). The benchmarks in LRA11 are used to test the value of various features of MATHSAT5 in an incremental setting[7].

[5] More generally, all the SAT-based simplification techniques which are not sound in an SMT context (such as e.g. the pure literal rule) must be switched off.

[6] Here, we are implicitly assuming that the SAT solver exposes an API similar to that of modern CDCL solvers such as e.g. MINISAT or LINGELING.

[7] The benchmarks in HSver could be also organized as incremental; however, the number of subsequent satisfiability queries is very low (two orders of magnitude lower than LRA11), and thus the results are not particularly informative.

Table 1. Results for MATHSAT5 with and without preprocessing on the BVuMEM and HSver benchmark classes.

Benchmark Family	Size	MATHSAT5				MATHSAT5$_{\text{PREPROCESSING}}$			
		#Solved	RT (sec)	#TO	#MO	#Solved	RT (sec)	#TO	#MO
brummayerbiere2	22	15	2218	5	2	**16**	**2014**	**6**	**0**
brummayerbiere	293	229	25698	64	0	**233**	**22620**	**60**	**0**
calc2	36	30	7855	6	0	**30**	**7301**	**6**	**0**
stp	40	26	2659	6	8	**27**	**3127**	**5**	**8**
HSver	279	260	6192	19	0	**279**	**2182**	**0**	**0**

MATHSAT configurations. In our experiments we have used the following versions of MATHSAT:

MATHSAT4: The latest version of MATHSAT4 (version 4.2.17).
MATHSAT5: The baseline MATHSAT5 configuration.
MATHSAT5$_{\text{PREPROCESSING}}$: MATHSAT5 with preprocessing enabled.
MATHSAT5$_{\text{CLEANELING}}$: MATHSAT5 using CLEANELING as a pluggable SAT solver.
MATHSAT$_{\text{MINISAT}}$: MATHSAT5 using MINISAT as a pluggable SAT solver.[8]

Experimental Set Up. All benchmarks were run on an xcore X5650 platform running Linux version 2.6.32, with a 32GB memory limit and a 20 minute time limit. In the tables, we use the following acronyms: RT for Runtime, TO for Time Out, MO for Memory Out.

Experiments. The intent of the first set of experiments is to evaluate the impact of SAT-level preprocessing. We focus on theories that cannot be directly reduced to pure SAT, showing that our approach is useful outside of pure $\mathcal{BV}$ problems. Table 1 shows the results of running MATHSAT5 both with and without preprocessing on the BVuMEM and HSver benchmarks. Figure 3 presents the corresponding scatter plots. In the BVuMEM benchmarks, the activation of the preprocessor allows MATHSAT5 to solve a higher number of instances. We notice that the activation of the preprocessor is not always positive, as it may result in time outs in cases solved without preprocessing (3 benchmarks). In terms of runtime, on the benchmarks solved in both cases, preprocessing yields a 15% In the HSver benchmarks, on the other hand, the positive effect of preprocessing is very evident, with 19 more instances solved, and a 2.8x speed up on average runtime. On single benchmarks, we notice an improvement of up to two orders of magnitude.

[8] Notice that, although both MINISAT and CLEANELING support SAT preprocessing, we had to turn it off when integrating them with MATHSAT5, since their SAT preprocessing procedures do not satisfy the requirements listed in §4.3 (see also §4.4).

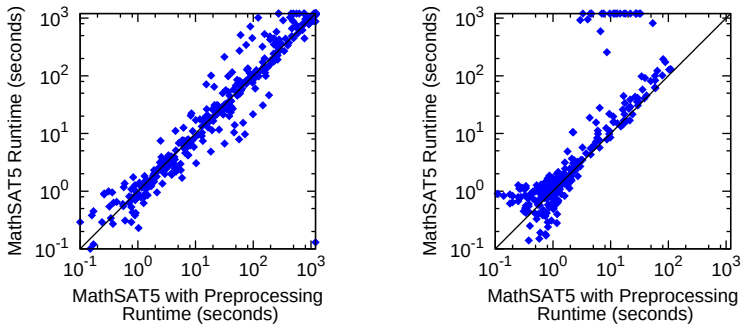

Fig. 3. Impact of preprocessing in the BVuMEM (left) and HSver (right) classes.

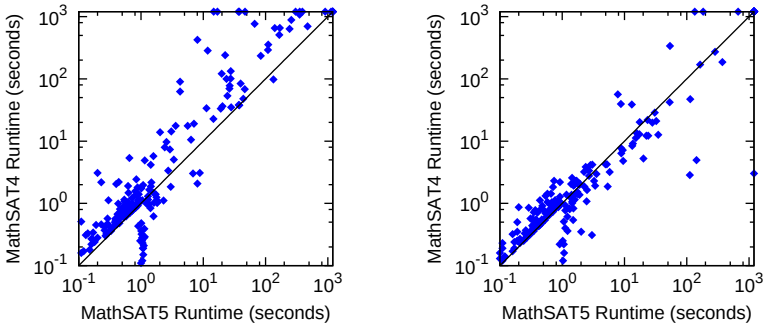

Fig. 4. MATHSAT5 versus MATHSAT4 on $\mathcal{EUF}$ (left) and $\mathcal{LA}(\mathbb{Q})$ (right).

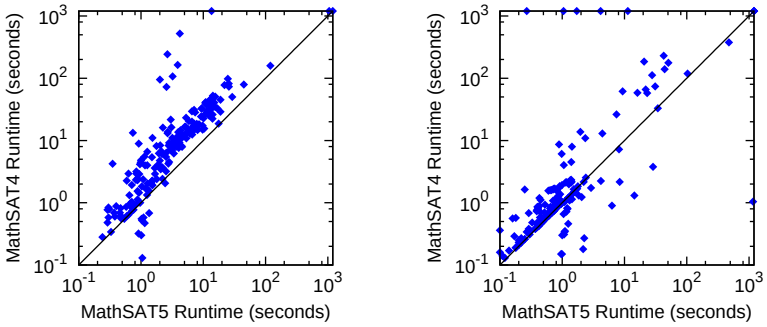

Fig. 5. MATHSAT5 versus MATHSAT4 on $\mathcal{LA}(\mathbb{Z})$ (left) and $\mathcal{BV}$ (right).

In the second set of experiments we compare MATHSAT5 against our previous solver MATHSAT4, using the COMP09 benchmarks. The results of the experiment are aggregated in Table 2, and displayed in scatter plots in Figures 4 and 5. From the data presented we can clearly conclude that MATHSAT5 outperforms MATHSAT4. We notice significant improvements in the $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Z})$ categories.

Table 2. A comparison between MATHSAT4 and MATHSAT5 on the COMP09 benchmarks.

Category	Size	MATHSAT4				MATHSAT5			
		#Solved	RT (sec)	#TO	#MO	#Solved	RT (sec)	#TO	#MO
$\mathcal{BV}$	200	192	1939	8	0	**197**	**2295**	**3**	**0**
$\mathcal{EUF}$	200	186	9317	14	0	**196**	**6232**	**4**	**0**
$\mathcal{LA}(\mathbb{Z})$	205	202	3985	3	0	**204**	**2205**	**1**	**0**
$\mathcal{LA}(\mathbb{Q})$	202	182	1588	20	0	**184**	**2816**	**18**	**0**

Table 3. A comparison between the MATHSAT5$_{\text{CLEANELING}}$, MATHSAT5$_{\text{MINISAT}}$ and MATHSAT5 solvers on the BVuMEM instances.

Benchmark Family	Size	MATHSAT5$_{\text{CLEANELING}}$				MATHSAT5$_{\text{MINISAT}}$				MATHSAT5			
		#Solved	RT (sec)	#TO	#MO	#Solved	RT (sec)	#TO	#MO	#Solved	RT (sec)	#TO	#MO
brummayerbiere2	22	12	709	8	2	**15**	**1831**	**5**	**2**	15	2218	5	2
brummayerbiere	293	164	23383	97	29	184	17044	97	12	**229**	**25698**	**64**	**0**
calc2	36	29	5852	7	0	**36**	**4183**	**0**	**0**	30	7855	6	0
stp	40	27	3595	5	8	**29**	**1765**	**3**	**8**	26	2659	6	8

In order to assess the pluggable SAT solver feature, we created to two versions of MATHSAT5 by integrating two external solvers[9]: MINISAT [21] and CLEANELING [18]. The cost of the integration turned out to be very moderate. This supports the claim that specialised SAT solver could be integrated and exploited successfully with a low initial effort.

Then, we compared these two solvers on the BVuMEM benchmarks. The results are detailed in Table 3. Compared to the version with our native solver (Table 1), the performance of the version with MINISAT is mixed: MATHSAT5$_{\text{MINISAT}}$ performs slightly better on three families, but much worse on the brummayerbiere family. The version with CLEANELING instead is inferior to our native solver. In general, SAT solvers and DPLL($\mathcal{T}$) SAT enumerators might have different requirements, so it's not obvious that a state-of-the-art SAT solver is always the best choice in DPLL($\mathcal{T}$). For example, the rapid restart policy used by modern SAT solvers might not be the best choice in SMT. Rebuilding the assignment stack after a restart is relatively cheap in pure SAT; however, in SMT it can be more expensive, since the theory solvers still need to perform consistency checks and provide deductions.

We also tested the version with pluggable solver on incremental benchmarks. Since CLEANELING does not support solving under assumptions, and thus cannot be used incrementally by MATHSAT5, we compared the performance of MATHSAT5 and MATHSAT5 using MINISAT on the LRA11 benchmarks set (Table 4). These problems are either bounded model checking (where the benchmark name contains "bmc"), or k-induction (name contains "ind") problems. Interestingly, k-inductions checks are much more efficiently solved by pure MATHSAT5, while the version using MINISAT handles bounded model checking instances much more efficiently. We are currently investigating the reasons for this difference.

[9] The code for the integration (see §4.4) is available from the web page of MATHSAT5 [33].

Table 4. A comparison between MATHSAT5$_{\text{MINISAT}}$ and MATHSAT5 on the LRA11 instances.

Benchmark	MATHSAT5$_{\text{MINISAT}}$		MATHSAT5	
	Reached bound	Runtime (sec)	Reached bound	Runtime (sec)
bmwlin_20_5_1.inter.bmc_k100	**101**	**25**	101	344
fisher_ring_20_3.inter.bmc_k100	**62**	**1200**	55	1200
dist_controller_15_3.inter.bmc_k100	76	1200	**93**	**1200**
rod_30_3.inter.bmc_k100	**101**	**66**	80	1200
fisher_star_20_3.inter.bmc_k100	**101**	**40**	101	367
rod_30_3.inter.ind_k100	27	1200	**45**	**1200**
mwlin_20_5_1.inter.ind_k100	47	1200	**133**	**1200**
fisher_star_20_3.inter.ind_k100	35	1200	**65**	**1200**
fisher_ring_20_3.inter.ind_k100	33	1200	**51**	**1200**
dist_controller_15_3.inter.ind_k100	31	1200	**69**	**1200**

In order to assess the strength of MATHSAT5 relative to the current state of the art (e.g. Boolector and Z3) we rely on the results of the 2011 and 2012 SMT-COMP. The version of MATHSAT5 presented in this paper is an extension, with new features, of the version which ran in those competitions. Thus the SMT-COMP results are relevant to this version. The competition results show that in non-incremental categories MATHSAT5 is generally competitive with other modern SMT solvers. In the incremental categories, it performs extremely well, winning many of them. Thus MATHSAT5 achieves its goal of being an efficient incremental solver, that supports a multitude of logics.

6 Applications

MATHSAT has been and is currently used in many research and industrial projects.

We have a long-standing collaboration with Intel FV group at Haifa, Israel, within the Intel- and SRC-funded BOWLING, WOLFLING and WOLF projects, in which MATHSAT has been used as backend engine for formal verification of RTL designs microcode [24]. In particular, a customized version of MATHSAT is currently integrated within the production version of Intel's microcode-verification suite, MICROFORMAL, and successfully used inside the company [24]. Another application in the verification of RTL is in the ForSyn [23] tool, where MATHSAT is the decision procedure used for checking the equivalence between RTL implementations and their high-level descriptions.

MATHSAT has been used as a backend in an extended version of the NuSMV model checker, called NuSMV3 [36]. NuSMV3 is a general synchronous extensions to the publicly available NuSMV2, where MATHSAT is used as a backend for SMT-based verification techniques. Among these, we mention bounded model checking, k-induction, and predicate abstraction. In these applications, the role of SMT is to provide a high level representation of the transition system. Various functionalities are exploited, including incremental reasoning, unsatisfiable core extraction, and interpolation.

The availability of MATHSAT has provided a basis for the extension of NuSMV to deal with analysis of hybrid systems. Hybrid systems are symbolically modeled in a language called HyDI, and specialized forms of verification [14, 13] strongly rely on the availability of advanced capabilities of MATHSAT. In the setting of hybrid systems verification, MATHSAT also supports the analysis of parametric timed automata [16].

The EuRailCheck project, funded by the European Railways Agency, relies on the MATHSAT-based requirements analysis capabilities [17].

The underlying verification capabilities provided by NUSMV3 are used in the ESA-funded projects COMPASS [5], AUTOGEF, FAME, and FOREVER, where complex aerospace systems are modeled in terms of hybrid automata.

MATHSAT is used as a backend for the analysis of temporal reasoning under uncertainty [12], within applications in the ESA-funded project IRONCAP.

An important class of applications of MATHSAT in software model checking. In particular, MATHSAT is integrated within the CPAchecker [3] and UFO [1] model checkers for sequential software, and within Kratos [14], a model checker for sequential and threaded software. Within this setting, MATHSAT supports the basic model checking steps (interpolation, predicate discovery, localization and post-image computation) by means of interpolation, unsatisfiable core extraction, and AllSMT. More recently, MATHSAT has been used as backend for an IC3-based approach to software model checking [9], and for parametric analysis of threaded programs [15].

7 Conclusions and Future Developments

In this paper we have presented the SMT tool MATHSAT5. In comparison to its predecessor MATHSAT4, substantial improvements have been made: in addition to significant improvements in efficiency, the key changes include extension to more theories, full support for incrementality, an incremental and SMT-aware preprocessor, and support to plug in third-party SAT solvers.

MATHSAT is a long-term project, and its development is ongoing. First, we plan a deeper investigation of SMT-aware preprocessing techniques, with the goal to make them available within a stand-alone functionality, so that to make MATHSAT work also as an effective *formula simplifier*. Second, we plan to investigate and implement *quantifier elimination* techniques for some of the theories of interest. We are also considering to investigate extensions to non-linear arithmetic.

A research direction we are currently pursuing is that of *Optimization Modulo Theories (OMT)*, which leverages SMT solving from *decision* to *optimization* level by finding models that *minimize* some given cost functions. Our previous work has produced variants of MATHSAT able to minimize cost functions on the pseudo-Boolean and $\mathcal{LA}(\mathbb{Q})$ domains respectively [8, 37]. Current and future work in this direction includes the porting of the OMT implementations of [8, 37] into the official MATHSAT5 version, and extensions to $\mathcal{LA}(\mathbb{Z})$ and $\mathcal{LA}(\mathbb{QZ})$ cost functions.

References

1. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In *Proc. CAV*, volume 7358 of *LNCS*. Springer, 2012.
2. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, chapter 26. IOS Press, 2009.
3. D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Proc. of CAV*, LNCS. Springer, 2011.
4. M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Theory Combination via Boolean Search. *Information and Computation*, 204(10):1493–1525, 2006.
5. M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Safety, Dependability and Performance Analysis of Extended AADL Models. *Comput. J.*, 54(5), 2011.
6. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *Proc. of CAV*, volume 5123 of *LNCS*. Springer, 2008.
7. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: A Comparative Analysis. *Annals of Mathematics and Artificial Intelligence.*, 55(1-2), 2009.
8. A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *TACAS*, volume 6015 of *LNCS*. Springer, 2010.
9. A. Cimatti and A. Griggio. Software Model Checking via IC3. In *Proc. CAV*, volume 7358 of *LNCS*. Springer, 2012.
10. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *ACM TOCL*, 12(1), 2010.
11. A. Cimatti, A. Griggio, and R. Sebastiani. Computing Small Unsatisfiable Cores in SAT Modulo Theories. *Journal of Artificial Intelligence Research, JAIR*, 40:701–728, April 2011.
12. A. Cimatti, A. Micheli, and M. Roveri. Solving Temporal Problems Using SMT: Strong Controllability. In M. Milano, editor, *Proc. of CP*. Springer LNCS, 2012.
13. A. Cimatti, S. Mover, and S. Tonetta. SMT-based Scenario Verification for Hybrid Systems. *Formal Methods in System Design*, 2012.
14. A. Cimatti, I. Narasamdya, and M. Roveri. Software Model Checking with Explicit Scheduler and Symbolic Threads. *Logical Methods in Computer Science*, 8(2), 2012.
15. A. Cimatti, I. Narasamdya, and M. Roveri. Verification of Parametric System Designs. In *Proc. FMCAD*. FMCAD, 2012.
16. A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In *IEEE Real-Time Systems Symposium*, 2008.
17. A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. Validation of Requirements for Hybrid Systems: a Formal Approach. *TOSEM*, 21(4), 2013.
18. CLEANELING. `http://fmv.jku.at/cleaneling/`.
19. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, 2006.
20. N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT*, volume 3569 of *LNCS*. Springer, 2005.
21. N. Eén and N. Sörensson. An Extensible SAT-solver. volume 2919 of *LNCS*. Springer, 2003.
22. N. Eén and N. Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.
23. ForSyn. `http://www.cs.utexas.edu/~sandip/projects/behavioral-synthesis/index.html`.

24. A. Franzén, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev. Applying SMT in symbolic execution of microcode. In *FMCAD*, pages 121–128, 2010.
25. The GNU Multi Precision Arithmetic Library. `http://gmplib.org`.
26. A. Goel, S. Krstić, and A. Fuchs. Deciding array formulas with frugal axiom instantiation. In *Proceedings of SMT'08/BPR'08*, pages 12–17, New York, NY, USA, 2008. ACM.
27. A. Griggio. *An Effective SMT Engine for Formal Verification*. PhD thesis, DISI - University of Trento, 2009.
28. A. Griggio. Effective word-level interpolation for software verification. In *FMCAD*, pages 28–36. FMCAD Inc., 2011.
29. A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation - JSAT*, 8:1–27, 2012.
30. A. Griggio, T. T. H. Le, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo linear integer arithmetic. *Logical Methods in Computer Science.*, 8(3), 2012.
31. L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding Floating-Point Logic with Systematic Abstraction. In *Proc. of FMCAD*, 2012. To Appear.
32. S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *Proc. CAV*, LNCS 4144. Springer, 2006.
33. MathSAT 5. `http://mathsat.fbk.eu/`.
34. A. Nadel, V. Ryvchin, and O. Strichman. Preprocessing in Incremental SAT. In *Proc. SAT*, volume 7317 of *LNCS*. Springer, 2012.
35. R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.
36. NuSMV 3. `https://es.fbk.eu/tools/nusmv3/`.
37. R. Sebastiani and S. Tomasi. Optimization in SMT with LA(Q) Cost Functions. In *IJCAR*, volume 7364 of *LNAI*, pages 484–498. Springer, July 2012.