Towards an Efficient Library for SAT: a Manifesto

Enrico Giunchiglia [1], Massimo Narizzano [1],

DIST, Università di Genova, Viale Causa, 13,16145 Genova, Italy

Armando Tacchella [2], Moshe Y. Vardi [2]

CS Dept., Rice Univeristy, 6100 Main St. MS 132, Houston, TX 77005-1892

Abstract

In recent years, propositional satisfiability (SAT) has become increasingly popular outside its traditional AI and automated reasoning niche. Applications from diverse fields require efficient and sophisticated SAT solvers that can be easily integrated into existing tools. In this paper we outline a path towards an efficient library for SAT that should meet the rising standards of research and applications. Our first milestone is SIM, a library for clausal SAT based on the well known Davis-Logemann-Loveland (DLL) method. SIM provides a choice of heuristics and optimizations that are found separately in most state-of-the-art DLL-based solvers, and extensive experimental evaluation confirms that SIM's generality does not hinder its performance.

---

1 Introduction

The issue of dealing effectively with the propositional satisfiability problem, SAT, is becoming more and more popular outside the traditional practice of AI and automated reasoning. Researchers and developers from diverse fields are attracted both by the existence of natural SAT encodings for many relevant problems, and by the availability of powerful SAT solvers. Systems like PROVER [Stå94], GRASP [SS96], SATZ [LA97], RELSAT [BS97] and SATO [ZS00], to mention only some, have been successfully used to build

[1] Partially supported by grants from MURST and Intel Corporation.
[2] Partially supported by NSF grants CCR-9700061 and CCR-9988322, BSF grant 9800096, and a grant from the Intel Corporation.

tools for planning [KS92], knowledge representation [GGT00], automatic theorem proving [CSGG00], and formal verification [BCCZ99].

In most applications the focus is not on dealing with specific SAT instances only, but on representing, manipulating and reasoning with propositional formulas in the widest sense. Most of the times, SAT solvers end up in being integrated into existing systems or used as the basis of more complex tools. It turns out that a practical, and by far the most popular, solution to integrate SAT technology into other systems is to use SAT solvers as "oracles", i.e., supply them with a SAT instance and ask them to solve it. All the pre- and post-processing routines are implemented and executed independently from the SAT solver, which is considered as a "black-box" in the terminology of [KS98]. The success of this approach is mainly due to the fact that state-of-the-art SAT solvers still lack a complete and well established interface, and most of them do not provide enough flexibility, support, and primitives to manipulate formulas. This is somehow frustrating, particularly considering that other families of propositional logic tools do not show these limitations. For instance, if we look at Ordered Binary Decision Diagrams (OBDDs) [Bry92], we see that practically all known implementations (see, e.g., CUDD [Som]) share the same well established interface to handle arbitrary propositional formulas. In addition, they provide a rich set of primitives including sophisticated ones like, e.g., formula substitution, restriction, and quantification.

Given the increasing importance of SAT in applications and the current state of the art, we single out two essential requirements for the next generation of SAT tools:

(1) a standard interface in the spirit of OBDD packages, and
(2) an open, modular and extensible design that does not sacrifice efficiency.

Requirement (1) fosters interchangeability between SAT tools and enables users to quickly leverage improvements in the current state of the art; requirement (2) ensures that future changes in core technology can be incorporated into SAT tools without extensive coding effort. We believe that SAT community and users will benefit more from reasonably effective application program interfaces (APIs) for SAT rather than super-efficient, but otherwise esoteric, SAT solvers.

In this regard, APIs can deliver a practical perspective to the issue of accurate and fair experimentation with SAT algorithms raised in [Hoo96]. Researchers can exploit APIs as an implementation platform, and benchmark their algorithms without getting results biased by different coding of basic services. Arguments against APIs often come from field users whose first concern is the ability to fine tune and adapt their systems for the application at hand. We think that such need can be accommodated by APIs, as long as we expose their internal interfaces. By exploiting low-level primitives, users can build their own high-level ones and custom fit the API for specific purposes. Of course, this exposes the users to the risk of misusing the API low-level primitives. Nevertheless, our experience with application domains where the focus is on the reuse of SAT technology (e.g., decision procedures for modal logics [Tac99]) clearly indicates that the ability to access the low-level mechanisms of the solver through a well established interface is worthwhile, for it enables users to implement better algorithms for existing high-level services without knowing the details of the internal data structures but only the primitives that manipulate them. In this way, the range of services available in the API can be effectively extended without extensive (re)coding effort which is required in systems that do not expose their internal interface. We further notice that the distinction between high- and low-level primitives has an added value that goes beyond the ability to custom fit the API. We believe that the availability of a low-level interface will indeed open the path to integrations among different approaches to SAT, like, for instance, the one proposed in [GYAG00].

A first answer to the some of the aforementioned issues is SIM (Satisfiability Internal Module). SIM was developed at the University of Genoa [GMTZ01], and designed as a library of heuristics and techniques for the satisfiability of propositional formulas in clausal normal form. SIM was aimed from the very beginning at providing SAT services to other applications, thus overcoming one of the major limitations of most present and past SAT solvers. SIM is based on the well known Davis-Logemann-Loveland (DLL) method [DLL62], whose implemented variants [SS96,LA97,BS97,ZS00] were shown to be among the best complete methods for tackling SAT. An extensive experimental evaluation done with SIM, confirmed that the integration of different approaches available in the literature is possible without significant performance loss. Indeed, SIM used as a solver performs as well as other state-of-the-art DLL-based SAT engines, and it provides combinations of heuristics and techniques that were not available before.

Many of our ideas on how to design an efficient API for SAT come from our experience in (re)using SIM and other SAT solvers in several applications:

- SAT-based decision procedures in modal logics, as an alternative to tableaux methods (see e.g. [GGT00,GT00]);
- evaluation of quantified Boolean formulas (QBF) as in [GNT01a];
- SAT-based planning with expressive action languages, see e.g. [CGT01];
- symbolic model checking, mainly as an alternative to OBDDs, as in [CFF$^+$01];
- SAT algorithms, heuristics and optimizations research as in [GMTZ01];

Currently, SIM's C++ prototype SIMO (SIM Object-oriented) is integrated in Thunder [CFF$^+$01], a model checker developed at Intel, while SIM is integrated in NuSMV [CCGR00], a model checker developed at IRST/ITC in collaboration with CMU. Both projects aimed at enhancing existing OBDD-based model checkers with additional capabilities along the lines of [BCCZ99]. SIM is also at the heart of the system QuBE [GNT01a], a solver for QBF whose performance qualifies it among the fastest state-of-the-art solvers currently

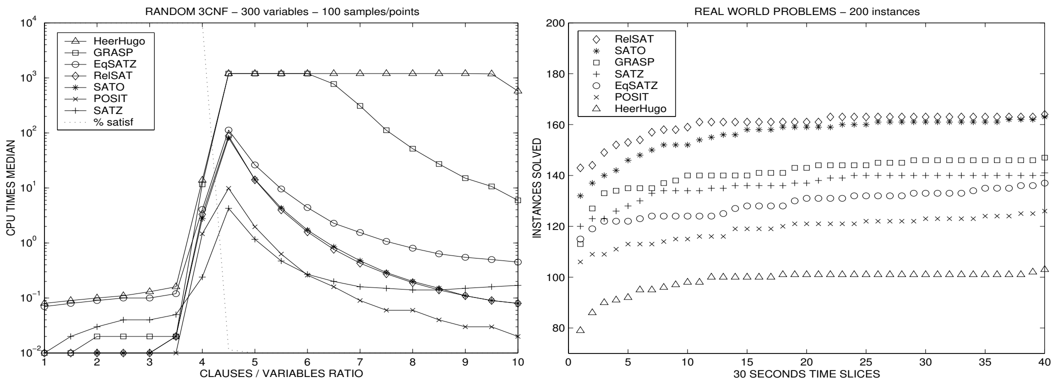

Fig. 1. State-of-the-art SAT solvers on randomly generated (left) and real world instances (right).

available (see [GNT01b] for an up-to-date comparison). Finally, SIM is also a component in the C-plan system [CGT01], a planner for very expressive action languages (see, e.g., [GL98]). All these projects provide us with a considerable source of design ideas for our next generation SAT library, SIM-API.

The paper is structured as follows. In Section 2 we give an overview of the state of the art in SAT. In Section 3 we discuss SIM's architecture and experimental results. In Section 4 we outline the design goals and the architecture of SIM-API, and in Section 5 we discuss about the expected advantages of SIM-API in SAT and SAT-related research projects. We end the paper in Section 6 with some concluding remarks and our agenda for future work.

2 State of the art in SAT

A preliminary step in the design of SIM was the assessment of the current state of the art in SAT. This required the identification of a suitable set of benchmarks, and the testing of – some of the – currently available SAT solvers on these benchmarks. Here we present just a brief summary of this activity. For a more detailed presentation of the topics in this Section, we refer to [GMTZ01].

SAT solvers are traditionally compared on "real world" problems arising in SAT applications and/or "random" problems generated according to some probability distribution. It is generally believed that instances belonging to the two categories have a different degree of "structuredness" and therefore different characteristics. While researchers are still trying to come up with a measure of structuredness in search problems (see e.g. [Wal99]), we pragmatically decided to get the picture as complete as possible by including both real world and randomly generated instances in our set of benchmarks. [3] In more

[3] SATLIB (`http://www.satlib.org/`) provides an extensive and up-to-date collection of SAT benchmarks.

detail:

- our test set of real world problems consists of 200 instances coming from well known benchmark sets in the SAT literature: given an initial target of 100 satisfiable formulas and 100 unsatisfiable ones, we included representatives coming from diverse domains that – if possible – have already been used for comparing SAT solvers;
- our test set of random formulas is generated with the fixed-clause-length model introduced in [FP83]: we ran experiments with different number of variables N and clauses L, and different clause lengths K; here we show the results for the setting $N = 300$ ($0 < N/L < 10$) and $K = 3$, which provides a reasonably hard benchmark for the solvers at hand.

In our experimental analysis, we restricted our attention to publicly available SAT solvers. Among these, we considered some of the most effective DLL-based ones. In particular, we picked POSIT ver 1.0 [Fre95], SATZ ver. 213 [LA97], EqSATZ [Li00], RelSAT ver. 2.00 [BS97], SATO ver 3.2 [ZS00] and GRASP ver. Feb. 2000 [SS96]. We also evaluated HeerHugo [GW00], the only publicly available implementation of the Stålmarck's method [Stå94]. Although an essentially arbitrary selection, these solvers are all state-of-the art, and they all have interesting features. For the purpose of this paper, the most relevant one is probably that every solver in our selection features a unique combination of heuristics and optimization techniques that were shown to yield its optimal performance. [4] All the solvers in our selection come as stand-alone tools, their only "interface" being command line switches and an input format complying with the DIMACS standard for clausal satisfiability [Dim]. With such format, the SAT instance is arranged as a matrix of integers, lines corresponding to clauses and numbers corresponding to literals, i.e, n ($-n$) stands for literal p_n ($\neg p_n$). Aside of this, we note that HeerHugo can read an extended propositional syntax, allowing for non-CNF formulas to be tackled directly. For the sake of uniformity, in our experimental analysis all the solvers are handed CNF instances only. We further note that none of the solvers can be linked as an application library, and since they provide only a limited interface, integrating them into other systems is generally not an easy task (see, e.g., [GGT00]).

Figure 1 shows the performances of POSIT, SATZ, EqSATZ, RelSAT, SATO, GRASP and HeerHugo on our test sets. The plot for the random tests in Figure 1 (left) is the usual one: the ratio N/L on the x-axis and the median CPU time on 100 instances on the – logarithmic scaled – y-axis (notice the curve in the background describing the transition of the satisfiability percentage from

[4] Experiments with the solvers run on several identical Pentium III, 600MHz, 128MBRAM. The execution of a system on a sample (be it random or real world) is stopped after 1200s of CPU times. All the solvers but SATO, which required a little tuning on random formulas, have been run in their default configuration.

100% to 0% for increasing values of N/L). The plot for the real-world tests in Figure 1 (right) shows on the y-axis the cumulative sum of samples solved by each solver within the amount of time on the x-axis (30s sampling intervals). As it can be seen, POSIT and SATZ are very effective on random tests, while RelSAT (with 164 samples solved) and SATO (with 163) are very effective on the real world ones. These data could have been expected given that POSIT and SATZ (resp. RelSAT and SATO) have been tuned to be effective on random (resp. real world) problems.

Looking at these results we immediately see that no SAT solver is outperforming the others on both random and real world problems. In other words, none of the solvers seems to be a general tool, i.e., each one was probably developed and tested with particular applications in mind. While the nature of SAT is currently challenging our efforts to come up with general purpose techniques, we believe that focusing our tools on specific instance classes implies loosing the scientific perspective in the investigation on SAT. Besides, it becomes almost impossible to relate the effectiveness of such tools – or their pitfalls – to algorithmic or coding factors. For instance, until we implemented SIM we were not able to asses whether the stunning performance of SATO and RelSAT on real world instances was due to their search heuristics, or to their optimizations, or to an excellent coding, or to a combination of all the factors. As remarked in [Hoo96] there is no point in competitive testing if, at the end of the day, we did not learn more about the nature of the underlying problem.

3 SIM as a platform for SAT

We now give a snapshot of SIM's internal architecture and experimental results: more details can be found in [GMTZ01]. SIM differs from the solvers mentioned in Section 2 in two important aspects. First, SIM is designed from the very beginning to be an application library that can be included by other tools. Interaction with SIM is thus possible via software primitives, rather than command-line switches and files. Second, SIM provides the user with a variety of heuristics and optimizations techniques that are found separately in most state-of-the-art SAT solvers. On the other hand, SIM cannot decide satisfiability of arbitrary propositional formulas, unless they are converted into clausal normal form. In this sense, SIM does not provide a more sophisticated service than most state-of-the-art SAT solvers.

SIM is built around the concept of *state* data object, first introduced by [Fre95] and improved in SIM to obtain the copy-less updating primitives herewith described. Intuitively, an instance of a state, i.e. a variable s of type state, encodes both a set of propositional clauses (constraints) and a truth assignment associated with the propositions occurring in the set. In the following, we say that a state is *inconsistent* if the truth assignment violates at least one constraint, and that it is *consistent* otherwise. A state is *satisfied* when all the constraints are satisfied by the truth assignment. A state is *satisfiable* if the truth assignment can be extended to yield a satisfied state. Clearly, an inconsistent state cannot be satisfiable. By *initial* state we denote the input SAT instance associated with the empty truth assignment. In our setting, a SAT instance is thus satisfiable if and only if the initial state is satisfiable. Besides the primitives to initialize a state, add clauses to it, access internal information, etc., SIM features two fundamental operations on a state s:

- Extend-Prop(s, l), which updates s by adding l to the truth assignment and propagating it to the set of clauses, and
- Retract-Prop(s, l), which, intuitively, reverts the effects of Extend-Prop.

Extend-Prop and Retract-Prop are still too fine-grained for a SAT search algorithm like a DLL variant to be built on top of them. Therefore SIM features a set of more coarse-grained actions on s described next.

Look-Ahead(s) deduces new truth assignments from s and propagates them, all in low-order polynomial time; in other words, Look-Ahead(s) keeps simplifying the state until a fix point is reached or an inconsistency arises.

Look-Back(s) tries to "repair" an inconsistent s and restore it to consistency; in other words, Look-Back keeps undoing truth assignments, until it reaches a point from which the search can continue, or it concludes that the initial state cannot be satisfied.

Heuristics(s) decides the next truth assignment and enforces it; the decision is taken by considering s and/or possibly some s' obtained from s by tentatively assigning truth values to literals.

It is easy to see that several search algorithms, including DLL variants, can be obtained from this basic template.

Solve(s)
1 *next* $\leftarrow$ LA
2 **repeat**
3   **case** *next* **of**
4   LA : *next* $\leftarrow$ Look-Ahead(s)
5   LB : *next* $\leftarrow$ Look-Back(s)
6   HR : *next* $\leftarrow$ Heuristics(s)
7 **until** *next* $\in$ {T, F}
8 **return** *next*

Following the conventions of [CLR98] we think of s as a pointer to a state (an instance of the state data type). This means that the actions Look-Ahead, Look-Back, and Heuristics all update the input state of Solve in various ways during the **repeat** ... **until** loop (lines 2-7). Each action modifies

s and returns the next action to be taken which is stored in *next* (lines 4-6 in the program). LA, LB, LA, T and F are the possible values taken by *next*, meaning that the next action is to be, respectively, LOOK-AHEAD, LOOK-BACK, HEURISTICS, or stop the loop. In the latter case, "*next*=T" means that s is satisfied, and "*next*=F" means that s cannot be satisfied. In the case of DLL the behavior is:

- start with LOOK-AHEAD: if the state is satisfied, the next action is T; if the state is inconsistent, the next action is LB; finally, if the state is consistent, the next action is HR;
- LOOK-BACK returns LA if the inconsistency is successfully repaired and the search can resume from a consistent state; it returns F if this is not possible, i.e., the initial state cannot be satisfied
- HEURISTICS always returns LA, unless it can perform some look-ahead itself, in which case it might also return T or LB.

SIM main algorithm is an implementation of the function SOLVE. SIM features some of the most popular kinds of look-ahead, look-back and heuristics and we are currently adding more. Currently, SIM's LOOK-AHEAD can perform the all-time classic unit clause detection and propagation, plus some other techniques like pure literal and failed literal detections; LOOK-BACK can be either chronological or enhanced with conflict resolution, thus including backjumping, and size- or relevance-bounded learning. Finally, SIM is featuring eight different search heuristics that can be variously combined with the above techniques to efficiently tackle different classes of problems as we show in Figure 2.

The experimental data presented in Figure 2 serves two purposes: show how SIM can be configured to fit the kind of input problem, and confirm that SIM's flexibility does not hinder its performance. In Figure 2, SimSATO means SIM using SATO's heuristics and optimizations, and similarly for SimRelSAT and SimSATZ. SIM numbers with these configurations are obtained on the very same test sets described in Section 2 and the plots are arranged in the same way.

Considering the top row of Figure 2, we notice that the plots on the random problems are similar to the ones in Section 2: SimSATZ performs better than SimReLSAT and SimSATO, these last two systems performing roughly in the same way; on real world problems, SimRelSAT being still the top performer, the picture changes a little bit: SimSATO does not quite perform as well as SimRelSAT. By looking at the correspondent state-of-the-art solvers in Figure 1, we see that this could not be expected, given that SATO and RelSAT on real world problems have roughly the same performance. Clearly, as we anticipated in Section 2, the comparison between SATO's and RelSAT's algorithms is obfuscated by other factors, e.g., coding, preprocessing and the like, which are sorted out by SIM.

Knowing that SIM is flexible and that it enables us to run fair experimen-

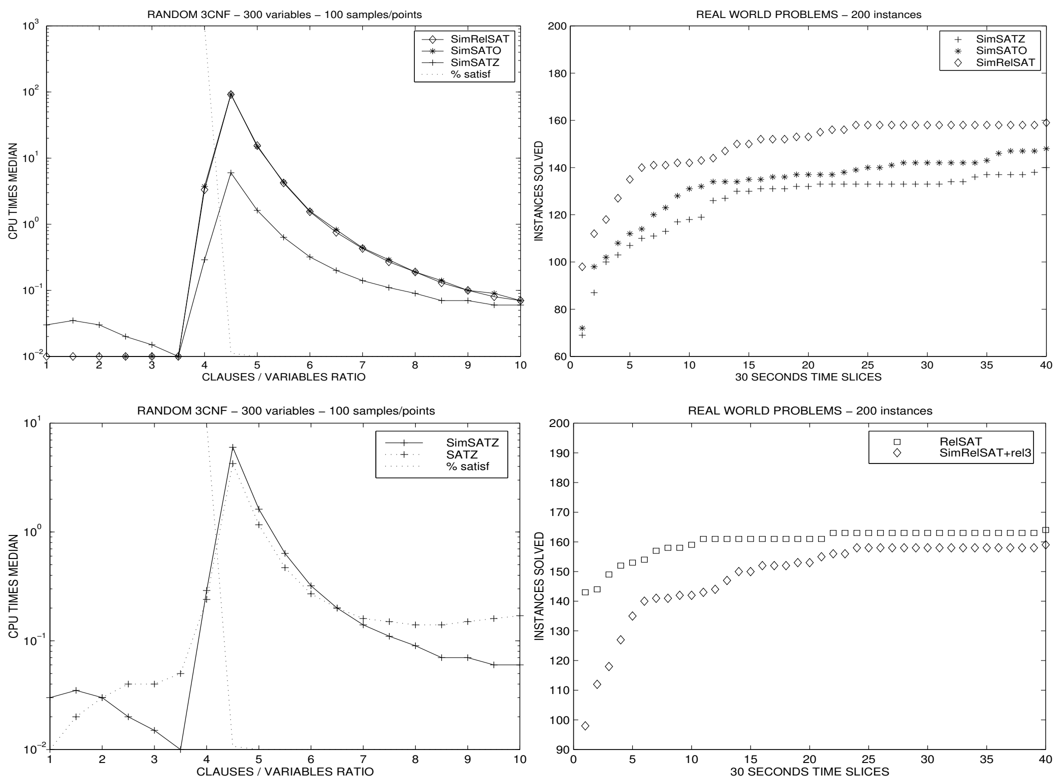

Fig. 2. SIM on randomly generated (left column) and real world instances (right column).

tal comparisons would be of little use if it performed badly. Considering the bottom row of Figure 2, we see that indeed this is not the case. Using comparable configurations, SIM is able to closely parallel the performance of SATZ on random problems and RelSAT on real world ones. We consider this as a first evidence that flexibility and thus generality can be obtained without sacrificing performances.

4 Designing SIM-API

Given the experience with SIM we now would like to build a software tool for SAT that matches requirements (1) and (2) outlined in Section 1. We recall that requirement (1) is about a standard and complete interface in the spirit of OBDD packages and (2) is about an open and modular architecture that does not sacrifice efficiency. Meeting these requirements is everything but an easy task, for (1) requires assessing – or even foreseeing – the needs in an expanding and diversifying users community, and (2) requires a significant effort in software engineering, together with basic data structures and algorithmic expertise.

The obvious question now is: how close is SIM to meeting these requirements?

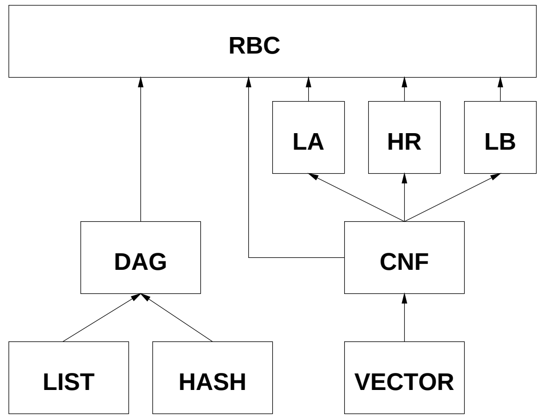

Fig. 3. An integration schema for SIM

Our answer is: close, but not close enough. As far as (1) is concerned, SIM's interface is thoroughly defined, and SIM can be used as an internal module – hence its name – in other tools including it as an application library. On the other hand, SIM complies to the only well known interface for clausal SAT, the DIMACS format for satisfiability. With the exception of basic tasks like choosing heuristics, optimizations, collecting statistics and the like, SIM's services that are exposed to the user are (i) adding clauses to the initial state and (ii) solving it. SIM does not allow to call, say, Look-Ahead independently from (ii), nor does it allow to manipulate the elements of the state in any ways other than (i). This might be good enough for pure SAT, but it is definitely not flexible enough for applications. For instance, in our experience with the integration of SAT solvers into model checking tools like Thunder [CFF$^+$01] and NuSMV [CCGR00], we soon realized that they needed the ability to manipulate arbitrary propositional formulas, using sophisticated primitives like substitution, restriction and quantification. In these projects, we ended up spending a lot of time in developing and optimizing packages that provided these services and that connected the model checkers to SIMO's and SIM's core primitives (i) and (ii). Concerning requirement (2), SIM still falls a little short as far as openness and modularity are concerned, although it definitely meets the requirement of effectiveness as shown in Section 3. We can say SIM is open, since it is made available in source code, it is written with rigid coding conventions, and it provides plenty of comment-generated documentation. On the other hand, SIM is still a single module and its basic data structures and algorithms are defined internally: there is no way to add a new heuristics or technique without knowing the details of SIM's data structure, and modifying – more or less locally – SIM's source code.

Therefore, even if we believe that SIM is already filling a gap in SAT research, we think it is time to further push the envelope. This is why we started with the design of a new tool that we named SIM-API. The purpose of SIM-API is to evolve SIM into what we believe the next generation of SAT engines. In the following, we outline the working hypotheses and the design choices that currently driving the development of SIM-API.

We wish to target SIM-API to the widest community of users interested in SAT, not just SAT researchers and/or specialists. Therefore, the external interface should be:

- capable of handling generic propositional syntax: this should satisfy the needs of most applications and should draw a sharp line between the high-level interface and the internal engine(s);
- capable of storing formulas effectively, i.e. providing a fast and compact representation: this essentially boils down to having a subformula sharing mechanisms implemented with efficient hashing schemas and a garbage collection;
- capable of various syntactic manipulations, e.g., substitutions, renaming, composition, as well as semantic ones, e.g., on-the-fly simplification, rewriting, normal forms: this should provide the users with simple means of building new formulas from existing ones and of (pre)processing them;
- capable of reasoning services not necessarily limited to SAT, e.g. enumerating assignments, finding small implicants, decomposing problems: all these services should be provided transparently from other internal mechanisms like, for instance, conversion to clauses.

SIM-API is also meant to be a research and development tool that evolves with time. As such, its implementation should be:

- designed around clearcut interfaces between internal modules: this should enable developers to access, replace or add internal components flawlessly and with minimal integration effort;
- well architected without sacrificing performances: this will require to carefully craft each component's data structures and algorithms;
- including the latest contributions to SAT research both in terms of engines, e.g. equivalence reasoning, stochastic methods, new heuristics, and in terms of additional services, e.g. incrementality, decomposition, etc.
- well documented, rigidly coded, and distributed in source code: this will speed up the evolution of the tool and foster collaboration on it.

As a glimpse into the future of SIM-API we present in Figure 3 the rational reconstruction of a combination between SIM and an RBC (Reduced Boolean Circuits [ABE00]) package that we used recently to integrate SIM into NuSMV [CCGR00]. RBC provides a representation of Boolean circuits (i.e., propositional formulas) as directed acyclic graphs, with additional properties that enforce unique representation of subformulas, simplification of the logical constants "true" and "false", and other normalization and rewriting rules (see [ABE00] for more details). The reason why RBC is relevant to this integration schema is that they provide a compact non-canonical representation for propositional formulas. Thus, the main purpose of the RBC package is to help SIM to interface with client algorithms by providing a layer of additional services, including, but not limited to, building propositional formulas with non-clausal syntax, substitution of subformulas, renaming of variables and conversion to clauses.

In Figure 3, RBC is providing client applications with a representation and manipulation infrastructure, while SIM is providing the basic reasoning services. In the figure, each box corresponds to a data type, and arrows denote "using" relationships: for instance, there is an arrow from LIST to DAG because the latter is building on the former. The picture is arranged in four layers (from bottom to top), corresponding to the basic modules LIST, HASH and VECTOR; to the intermediate support modules DAG, CNF; to the "semantically aware" modules Look-Ahead, Look-Back, HeuRistics; and, finally, to the interface level, currently RBC. Although still preliminary in many ways, the combination of SIM and RBC outlined in Figure 3 is providing us with the implementation groundwork and some architectural solutions that we are considering for the final outline of SIM-API.

5 SIM, SIM-API and where you can put them

In this section we review some applications of SAT technology spanning across the fields of logic, artificial intelligence and formal verification. Our considerations are based on our direct experiences with the (re)use of SIM –as well as other SAT solvers– to carry out various propositional reasoning tasks. The goal of this discussion is to show the potential benefit to these applications of reengineering SIM to become an API along the lines discussed in Section 4. For each application, we give some context motivating the choice of SAT as the underlying technology, as well as the current status of SAT-based tools. Then, we contrast the cost effectiveness of the "black-box" approach to the integration of SAT solvers with the expanded set of possibilities offered by SIM-API.

5.1 SAT algorithms, heuristics and optimizations

In [GMTZ01] an extensive experimental investigation is carried out, focusing on state-of-the-art search heuristics and optimization techniques in SAT. The key feature of the investigation is to have all the heuristics and techniques implemented in SIM, i.e., on top of the same basic data structures and services. This eliminates, or at least strongly attenuates, the bias associated with using different implementations of the basic components of the solver. Such bias is indeed unavoidable when comparing heuristics and optimizations across different solvers from different authors. Thanks to SIM, we are able to get a clear picture about the relative effectiveness of different search strategies.

The contribution of an API to this line of research can be manifold. First, and most important of all, an API architecture will allow researchers interested in SAT to have ideas implemented quickly into an efficient and tested framework. This will be possible, even with little or no knowledge of the "black magic" that is usually associated with engineering efficient components in SAT solvers. Therefore we believe that SIM-API has the potential to address the issue raised by Hooker in [Hoo96]: "Competitive testing tells us which algorithm is faster but not why. Because it requires polished code, it consumes time and energy that could be spent doing more experiments".

Second, although a substantial part of the ongoing research in propositional logic focuses on finding clever ways to satisfy/refute formulas, there are many other problems related to propositional reasoning that are amenable to algorithmic solutions and have practical relevance, such as counting the number of models, finding prime implicants, dealing with constraints other than clauses. These are all cases in which algorithms require implementations at least as sophisticated as the ones found in SAT solvers. On the other hand, recasting existing SAT solvers to solve some of the above mentioned problems is in general not feasible without rewriting at least part of the SAT engine, and this in turn requires a deep knowledge of the engine's code. Having an API will instead facilitate using SAT technology in a broader way, since users can now change the way the basic component interact and code (re)writing can be limited to the main reasoning algorithms.

Finally, problems are sometimes best solved with a hybrid approach, i.e., one that uses a combination of different search algorithms, heuristics and techniques to attack the problem at hand. Examples are combination between stochastic and complete search algorithms as in [MSG98,GSCK00], depth first and breadth first [GYAG00], or different heuristics and techniques on different subproblems [Li00]. The availability of crisply defined interfaces between the modules and the ability to combine and extend them, makes it possible to use the API's components in different and original ways.

5.2 SAT-based symbolic model checking

Symbolic model checking [BCM92,McM93] is a well established technique for formal verification of reactive systems such as hardware circuits and protocols. In symbolic model checking the system is modeled as a finite state machine (FSM) and the specification is expressed in temporal logic. The representation of the FSM is not explicit, but is provided as a Boolean encoding: this technique enables handling systems of industrial size. Ordered Binary Decision Diagrams [Bry92] (OBDDs), a canonical form for Boolean expressions, have traditionally been used as the underlying representation for symbolic model checkers [McM93]. Recently it has been shown how SAT solvers can be used instead of OBDDs in some interesting domains related to formal verification, and SAT solvers are gaining more and more importance also in this field. In [BCCZ99] the notion of bounded symbolic model checking (BMC) is introduced, and SAT solvers are showed to be capable of finding counterexamples for safety properties much faster than competing OBDD-based tools.

As we mentioned in section 1, the C++ prototype SIMO is integrated into Intel's SAT-based model checker Thunder, which performs BMC on top of SAT solvers (see [CFF$^+$01]). We note that SIMO also includes new heuristics that are specially tuned for the BMC problem domain. As reported in [CFF$^+$01] Thunder/SIMO achieves impressive capacity and productivity for BMC. Real designs, taken from Intels Pentium4, with over 1000 model variables were validated using the default tool settings and without manual tuning.

There are two main issues that this project presented to us. First, SIMO's special purpose heuristics for BMC had to be hard-coded into the solver and an author's expertise was required to do this and would be required to further update and modify the code. Second, SIMO's limited interface (DIMACS format) compelled the developers of Thunder to build and optimize a set of routines to perform conversion to clauses, which turned out to be an extremely time-consuming effort, particularly in the optimization phase. In the continuing effort to meet market pressure, formal verification needs to manage designs of ever increasing size and complexity. This calls for instruments that are easily tunable by the developers (or even the users) of the verification tools, rather than by the developers of the underlying SAT technology only. Moreover, the developers of formal verification tools should be able to concentrate on algorithmic issues rather than the optimization of low-level services such as conversion to clauses. The modularity and the definition of internal interfaces among the components in SIM-API should allow power users to reach the required level of customizability for SAT tools, with the additional benefit of a common interface towards client applications and a full set of services built-in the package.

5.3 Quantified Boolean formulas evaluation

The implementation of effective reasoning tools for deciding the satisfiability of Quantified Boolean Formulas (QBFs) is an important research issue in Artificial Intelligence. Many reasoning tasks involving abduction [EGL97], reasoning about knowledge [FHMV95], non monotonic reasoning [Cad95], are PSPACE-complete reasoning problems and are reducible in polynomial time to the problem of determining the satisfaction of a QBF.

It is along this line of research, that in [GNT01a] the system QuBE was presented. Like other QBF procedures (see, e.g., [CSGG00,Rin99,FMS00]), QuBE's basic algorithm is a generalization of the DLL method for SAT. Moreover, QuBE features several search heuristics, simplification techniques and optimizations like *trivial truth* [CSGG00] and *backjumping* [GNT01a]. The natural choice was to implement QuBE on top of SIM or, to be more precise, extend and modify SIM's basic data structures and primitives to deal with the problem of QBF evaluation. This allowed leveraging the current state of the art in SAT: for instance, the backjumping procedure implemented in QuBE [GNT01a] is a generalization of the conflict-direct backjumping procedure as implemented in SAT solvers.

Although QuBE shares with SIM many common components it is hardly possible to single out SIM's modules inside QUBE's implementation. This is clearly a problem when it comes to integrate SIM improvements into QuBE. For example, SIM is currently being extended to include equivalence reasoning techniques along the lines of [Li00]. The amount of code that is required to keep track of equivalence classes among variables and exploit them in the search process is indeed considerable, as well as the effort to test and optimize such machinery. Nevertheless, if we were to implement the same techniques into QuBE now, we could hardly reuse any of the code developed for SIM, and we would have to rewrite or modify parts of QuBE's basic services and data structures. On the other hand, in an API architecture, the modules that QuBE relies on could be updated transparently and improvements to SIM could be made available into QuBE with little or no effort.

5.4 Modal logics

In the seminal paper by Giunchiglia and Sebastiani [GS96], the authors described how to build a decision procedure for the modal logic K on top of the DLL method. Remarkably, it was the first time that a method different from the ubiquitous tableaux was proposed and implemented in a modal decision procedure. The link between SAT technology and the practice of modal logics has been exploited later in [GGST98,GGT00], where implementations of decision procedures were obtained by reusing state-of-the-art clausal SAT solvers. BOEHM solver [BKB92] was used to get KsatC implementation in [GGST98], and SATO is at the heart of the system platform *SAT [GGT00,Tac99].

Despite the amount of research carried over the years, the issue of whether the SAT-based approach is an effective way to build systems for modal reasoning is still a controversial one. In [GGT00] it is pointed out that (re)using existing SAT solvers speeded up the implementation of systems for modal reasoning and leveraged the latest improvements in technology. Nevertheless, such advantages come at the high price of "black-box" interfacing with pieces of software that are designed as stand-alone decision procedures and can rarely be tuned outside their standard range of features. The SAT-based approach was also challenged on its own grounds by the availability of fast tableaux-based [Hor97,PS98] and resolution-based [HS97] modal decision procedures. In some of these contributions, it was argued that SAT-based procedures had disadvantages that could limit their applicability because:

- the separation between the propositional and modal components is artificial and makes optimizations difficult,
- there can be limits on the expressiveness of the logics that are dealt with, and
- a fast propositional reasoner is not necessarily so in the context of modal logics.

We believe that these arguments do not stem from an intrinsic conceptual weakness of the SAT-based approach, but rather from the weaknesses of state-of-the-art SAT solvers that we discussed in previous sections. In particular:

- the separation between propositional and modal components is a consequence of the the "black-box" approach to integration;
- the modifications to adapt SAT solvers to various kind of modal logics are conceptually easy, but hard to get in practice without massive code rewriting;
- SAT solvers feature very powerful heuristics and search techniques that are highly tuned towards etiher finding satisfying assignments or refuting formulas very quickly, but the very same heuristics and techniques may not be so effective when the end goal is modal reasoning instead of pure SAT.

In other words, with current state-of-the-art SAT solvers, we are limited in our ability to get SAT technology off the shelf, and this turns out to be the main limiting factor for further development of the SAT-based approach in modal logics.

One of the main reasons behind SIM's development was to provide us with SAT technology that we could leverage in our modal decision procedures: since we developed SIM, we could modify it with considerably less effort than any other SAT solver. We are now pushing this further with the development of SIM-API: the plan is to build the next version of *SAT by using exactly those features that are made available to users. In this sense, being the developers of the API should not give us any "unfair advantage". By keeping in mind Figure 3 and the design goals outlined in Section 4, we remark the following:

- we can use SIM-API interface either by extending it to deal with modal logics, or by using the primitives that lie underneath the interface to build a new one specialized for modal logics: in both cases we save time for coding formula representations;
- formulas can be converted to CNF using API's primitives and stored using its internal data structures: this part had to be developed from scratch both in KsatC [GGST98] and in *SAT [Tac99].
- we can inherit various propositional functionalities from the API, and extend them to modal logics: for instance we can write a modal conflict-directed backtracking and use it to complement the one provided by the API;
- we can use the basic data structures, i.e., LIST and HASH and the like,

to build more complex types in *SAT, e.g., a cache module like the one described in [GT00]

In the realm of modal reasoning, we expect SIM-API to remove the limiting factors present in the current state-of-the-art SAT solvers and to enable us, as well as other developers, to fully exploit the advantages of the SAT-based approach.

6 Conclusions

In this paper we presented an up-to-date picture of the current state of the art in SAT technology, focusing on DLL-based solvers and on our direct experience in developing and (re)using them. In more details:

- by building SIM and RBC, we progressed towards an efficient and easy-to-use framework and we investigated the effectiveness of several heuristics and optimizations in the context of several applications.
- by (re)using state-of-the-art SAT solvers, we collected evidence on the effectiveness of different search strategies in different application domains, and we spotted the problems that make SAT solvers hard to reuse as components of more complex systems;

These experiences are now leading us towards the development of SIM-API, an efficient, open and modular framework for SAT and SAT-based reasoning. Our agenda on SIM-API encompasses the following items:

- give the final shape to the software architecture using state-of-the art software design methodology;
- flesh out the modules by carefully crafting data structures and components according to the experience with SIMO, SIM and RBC;
- deploy the API in the areas cited in Section 5 (and possibly other ones) to test and improve its applicability on the field.

In the end, our effort should provide the research community with a tool that can be used to leverage the latest improvements in SAT technology and pursue new research directions in propositional logic.

References

[ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic Reachability Analisys Based on SAT-Solvers. In *Proceedings of TACAS*, volume 1785 of *LNCS*, pages 411–425. Springer Verlag, 2000.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of TACAS*, volume 1579 of *LNCS*, pages

193–207. Springer Verlag, 1999.

[BCM92] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic Model Checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BKB92] M. Buro and H. Kleine-Büning. Report on a SAT competition. Technical Report 110, University of Paderborn, November 1992.

[Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BS97] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc. of AAAI*, pages 203–208. AAAI Press, 1997.

[Cad95] M. Cadoli. *Tractable Reasoning in Artificial Intelligence*. Number 941 in LNAI. Springer Verlag, 1995.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CFF+01] F. Copty, L. Fix, Ranan Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proc. of CAV*, LNCS. Springer Verlag, 2001. To appear.

[CGT01] C. Castellini, E. Giunchiglia, and A. Tacchella. SAT-based planning in complex domains: Concurrency, constraints and nondeterminism, 2001. Unpublished manuscript.

[CLR98] T. H. Cormen, C. E. Leiserson, and R. R. Rivest. *Introduction to Algorithms*. MIT Press, 1998.

[CSGG00] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and its Experimental Evaluation. In *Highlights of Satisfiability Research in the Year 2000*. IOS Press, 2000.

[Dim] The Second Dimacs Challenge: Satisfiability Suggested Format. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7):394–397, 1962.

[EGL97] T. Eiter, G. Gottlob, and N. Leone. Semantics and Complexity of Abduction from Default Theories. *Artificial Intelligence*, (1-2)(90):177–223, 1997.

[FHMV95] R. Fagin, J.Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*. MIT Press, 1995.

[FMS00] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proc. of AAAI*, 2000.

[FP83] J. Franco and M. Paull. Probabilistic analysis of the Davis-Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.

[Fre95] J. W. Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, 1995.

[GGST98] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. More evaluation of decision procedures for modal logics. In *Proc. of KR*. Morgan-Kauffmann, 1998.

[GGT00] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT-Based Decision Procedures for Classical Modal Logics. *Highlights of Satisfiability Research in the Year 2000*, 2000.

[GL98] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. of AAAI.* AAAI Press, 1998.

[GMTZ01] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. of IJCAR*, LNCS. Springer Verlag, 2001. To appear.

[GNT01a] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In *Proc. of IJCAI*, 2001. To appear.

[GNT01b] E. Giunchiglia, M. Narizzano, and A. Tacchella. On the effectiveness of backjumping and trivial truth in quantified boolean formulas satisfiability. In *IJCAR workshop on Theory and Application of Quantified Boolean Formulas*, 2001.

[GS96] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. of CADE*, LNAI. Springer Verlag, 1996.

[GSCK00] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. In *Highlights of Satisfiability Research in the Year 2000.* IOS Press, 2000.

[GT00] E. Giunchiglia and A. Tacchella. A Size-bounded Subset-matching cache for Satisfiability in Modal Logics. *Annals of Mathematics and Artificial Intelligence*, 2000. To appear.

[GW00] J. F. Groote and J. P. Warners. The propositional formula checker heerhugo. In *Highlights of Satisfiability Research in the Year 2000.* IOS Press, 2000.

[GYAG00] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-Based Image Computation with Application in Reachability Analisys. In *Proceedings of FMCAD*, volume 1954 of *LNCS*, pages 354–371. Springer Verlag, 2000.

[Hoo96] J. N. Hooker. Testing Heuristics: We Have It All Wrong. *Journal of Heuristics*, 1:33–42, 1996.

[Hor97] I. Horrocks. *Optimizing Tableaux Decision Procedures for Description Logics.* PhD thesis, University of Manchester, 1997.

[HS97] U. Hustadt and R.A. Schmidt. On evaluating decision procedures for modal logic. In *Proc. of IJCAI*, 1997.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of ECAI*, pages 359–363, 1992.

[KS98] H. Kautz and B. Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, 1998.

[LA97] C. M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proc. of IJCAI*, pages 366–371. Morgan-Kauffmann, 1997.

[Li00] C. M. Li. Integrating Equivalency Reasoning into Davis-Putnam Procedure. In *Proc. of AAAI*. AAAI Press, 2000.

[McM93] K.L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[MSG98] B. Mazure, L. Sais, and É. Grégoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22:319–331, 1998.

[PS98] P. F. Patel-Schneider. DLP System Description. In *Collected Papers from the International Description Logics Workshop (DL'98)*. CEUR Workshop Proceedings, 1998.

[Rin99] J. T. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proc. of IJCAI*, pages 1192–1197, 1999.

[Som] F. Somenzi. CUDD: CU Decision Diagram Package. Release 2.3.0.

[SS96] J. P. M. Silva and K. A. Sakallah. GRASP - A new Search Algorithm for Satisfiability. Technical report, University of Michigan, 1996.

[Stå94] G. Stålmarck. System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated From Boolean Formula, 1994. United States Patent, No. 5276897.

[Tac99] A. Tacchella. *SAT system description. In *Collected Papers from the International Description Logics Workshop (DL'99)*. CEUR Workshop Proceedings, 1999.

[Wal99] T. Walsh. Search in a Small World. In *Proc. of the 16th International Joint Conference on Artificial Intelligence (IJCAI '99)*, 1999.

[ZS00] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam Method. In *Highlights of Satisfiability Research in the Year 2000*. IOS Press, 2000.