The OKlibrary: A generative research platform for (generalised) SAT solving

by

Oliver Kullmann

Report # CSR 1-2008
The OKlibrary: A generative research platform for (generalised) SAT solving

Oliver Kullmann*
Computer Science Department
Swansea University
Swansea, SA2 8PP, UK
email: O.Kullmann@Swansea.ac.uk
http://cs.swan.ac.uk/~csoliver

January 18, 2008

Abstract
The OKlibrary is introduced, an active library supporting research and development in the area of generalised SAT-solving, available at http://www.ok-sat-library.org. We discuss motivation and architecture of the library, and outline its current extend. A key idea is the notion of “generalised satisfiability problems” as a generalisation of SAT towards CSP, and we discuss the basic ideas.

1 Introduction
In this paper I wish to introduce the “OKlibrary” to the SAT-world. The “alpha release” is scheduled with version 0.2.1 (in February 2008), the “beta release” with version 0.2.3 (around June 2008). At this time the focus is on foundational work, until the full public release with version 0.3 (the end of 2008). The library is open-source, licensed under the GPLv3 (which includes patent-protection which is needed especially since the library as a “holistic library” includes also research plans). The platform is available only for Linux: We put strong emphasise on standards w.r.t. programming languages, however as an active library the OKlibrary demands a strong build system, and it includes many other open-source projects, and this is only manageable for us under Linux.

1.1 The need for unification
The history of SAT (algorithms) could be divided (roughly) into 6 periods:

1. Beginning in the late 19th century, mechanical engines for finding all solutions to a boolean equation were built, but apparently these machines had no real practical use.

---

*Partially supported by EPSRC Grant GR/S58393/01
2. Starting in the thirties of the 20th century, circuit design and minimisation was the core focus.

3. In the fifties, algorithms for predicate logic with special consideration of the propositional part appeared.

4. After this “prehistory”, “modern SAT” started with the NP-completeness of SAT in the seventies, with a shift in focus now on proving lower bounds.

5. The eighties and nineties have seen an explosion in interest in random problems and the solution via local search methods.

6. The basic algorithm for SAT, the backtracking algorithm, finally has been strengthened, beginning in the nineties, by the incorporation of inference and learning (somewhat similar to earlier developments in the field of constraint satisfaction problems, but from a different point of view); especially large industrial applications enjoyed a boost since the beginning of this century.

Applied SAT solving has seen a strong development, typically in the area of formal methods where a SAT solver is used as an underlying “engine”; however other applications, for example in computational biology, now become also more common. It can be said that SAT solving definitely has changed the landscape of these application areas (some speak of “revolutions”). Now the momentum of the sixth period is running out — there is still some progress in implementation details and especially regarding the integration of SAT solving into complex applications, but progress regarding the core algorithms is stalled. We hope that the OKlibrary can help to start the seventh period, developing new fundamental methods for SAT solving, based on theoretical advances integrating SAT tightly with combinatorics and algebra, mathematical analysis and probability theory, and supported by a general and generative (open source) research and programming environment, strong enough to integrate many divers works of theoreticians and practitioners over the next 10-20 years.

What seems to me most characteristic for the current state of affairs are several dividing lines dominating the landscape, between areas which seem close, but yet withstood all attempts for unification:

1. Research on “intelligent”, complex algorithms, either considering polynomial-time algorithms for special classes, or deriving improved upper bounds, yet had no direct influence on practical SAT solving.

2. SAT and CSP attack similar problems from very different angles:

   (a) CSP concentrates on methods for (abstract) problem formulations and on exploiting special structures through incorporation of specialised (local) inference algorithms.

   (b) SAT exploits the concrete representation of the problem, and focuses on global algorithms based on statistical sensing of problem structure (global algorithms are enabled by the uniform representation, while modularisation as in CSP hinders “global thinking”, since the modules show different complex behaviours).

3. SAT algorithms are split into three areas without much overlap:
(a) On satisfiable random instances, local search algorithms (including their relatives coming from statistical physics) dominate, since here the underlying structures regarding inference are rather weak, while these solvers by their nature avoid the hard unsatisfiable sub-instances (otherwise obtained by splitting the problem), and the attracting forces of satisfying assignments are strong enough.

(b) On unsatisfiable random instances and many combinatorial instances, look-ahead algorithms dominate, relying on their fine-tuned inference schemes and focus on the worst-case (not hoping for good luck, but performing hard work to weaken the exponential behaviour).

(c) Finally, on large instances from applications like circuit design and also on several combinatorial problem classes, conflict-driven solver dominate due to their aggressive search for weak spots in the problem, and the fundamental break-through of integrating the search-organisation in the problem itself (via clause-learning).

Means for overcoming these three separations, supported by the OKlibrary, I see in the following approaches:

1. “Intelligent methods” need a strong interaction with combinatorics. The two main lines of attacks seem to be
   - Graph and hypergraph decomposition, with strong support by the (theoretical) CSP community and finite model theory.
   - The theory of autarky systems considers, orthogonal to the fields close to (hyper)graph decompositions, different combinatorial structures related to partial assignments, and seeks a unified general theory of “satisfiability phenomena”.

   Linear programming and generalisations have strong interactions with the above two approaches.

2. Via a generalised “topological” framework SAT and CSP are combined into a single framework of generalised satisfiability problems, where partial assignments are of fundamental importance (different from CSP), and where SAT algorithms serve as the basis of abstract data types, while CSP algorithms add local intelligence.

3. The combination of the three disjoint SAT-solver paradigms is envisaged by exploiting the duality of autarky and resolution (see [6]), establishing in a generalised sense a “primal-dual” scheme, where clause learning together with resolution establishes a “dual point of view” to the primal point of view given by partial-assignments, inference and autarkies.

SAT research has gained much from its practical side, and I regard unity of theory and practice as essential. The development lines outlined above do not need just a single library (as flexible and generic as it might be), but an integrated research environment (IRE), where abstract mathematical ideas can be combined with heuristical ideas on implementations: Such an environment must give support for diverse programming activities of many individuals over the next, say, 10 years — and this is what the OKlibrary seeks to establish.
1.2 The need for an “integrated research environment”

The original idea for the OKlibrary was a (generative) library for implementing “efficient SAT solvers” (which can compete in a SAT competition). This has been broadened to an “integrated research environment” (IRE) for SAT.

1. I believe the impact of complex and sophisticated methods will become much stronger in the future; and only with them the current stagnation of progress in SAT solving can be overcome. Thus the emphasise of the OKlibrary on complex “mathematical” algorithms.

2. This new research orientation comes together with a whole new “mind-set”: The main “clients” of the OKlibrary are now researchers (theoretical and practical), and researchers have a broader approach than “sheer programmers” wishing to assemble a new program out of given components.

Programmers just wishing to write a “sleek SAT solver” (with the chance of winning a competition) are (currently) not attracted by abstract methods, and (at least at this time) a generic implementation of a successful solver like Minisat (see the current SAT competitions) will have a time overhead (which at this time is unacceptable for such programmers). So what is needed, an abstract and “full” approach, must (at this time) focus on the diverse need of researchers (while the focused delivery of high-efficiency components will also be part of the OKlibrary, but this needs the next round of development).

Though researchers in logic, structures, algorithms, satisfiability and constraint satisfaction have a better grasp on abstraction (and for applying complex ideas and algorithms a generative library is most needed), these researchers are lacking in my opinion the methodology to bring their research into “real existence”: According to my experience, a “sociological” reason for the failure of complex methods to influence practical SAT solving lies in the following typical development process:

- The starting point is some fundamental idea.
- Since most researchers cannot implement it themselves, a PhD student or a research assistant is employed to do the job.
- This effort, creating a dedicated program with several hard-coded switches, halts after a few years, showing “promising results” which are “yet not competitive with the highly optimised existing solvers”.
- The PhD students and research assistants leave. And to put some postdoc on the project might be too risky for their scientific career.
- The code produced is not maintainable, and gets lost.

So, all efforts to bring sophisticated algorithms into practical SAT solving have failed until now. I believe this is, besides the theoretical foundations not understood yet, caused by the insufficient software engineering employed by such projects, since only a few combinations of methods can be hard-coded, experimentation takes too much time, and the development can not be sustained for a longer period. The OKlibrary seeks to change this picture by enabling the production of abstract and reusable algorithmic components which can be understood on a high level, and where experimentations with literally thousands of methods, evolving over time and tested on a multitude of test sets and benchmarks, can be performed, employing
the joined efforts of many researchers over many years. This scope of the library means that a novel infrastructure has to be build, which is given by the **OKplatform** as a “holistic” and “active” library, where all elements of the development process are fully integrated into the system itself — the user gets exactly the same power (by getting a fully functional “clone”) as the library creators themselves:

1. Testing is not left to the back-office, but “higher-level” test-functions are provided with the library itself.

2. Performance measurement (from measuring the performance of small components to running solvers on benchmarks in distributed environments) also comes with the platform.

3. Creating distributions and its infrastructure (like web pages and mailing-lists) again is integrated into the platform, available for every user, so that every user can create his own research network.

4. Last, but not least, full version control comes with the platform.

The traditional centralised (open source) software development model is unrealistic for the development of complex algorithmic methods, especially in the context of research, where researchers first need to develop and publish their new methods before they can be made generally available. So the **OKplatform** gives researchers a platform for organising their research first only within their group, but with the same methods as the developers of the **OKlibrary** have at hand, until their efforts might be integrated with the main library.

### 1.3 Comparison with other libraries

[2] has been influential for the early development of SAT libraries. It seems that the libraries developed by the authors (SIMO as the most prominent example) have taken a more commercial turn and are not available as open-source. [2] has strong connections to industrial applications, while the **OKlibrary** has yet stronger connections to academic research. Definitely shared is the concern for reusable components, however [2] requests a “standard interface in the spirit of OBDD packages”, which is not sought by the **OKlibrary**, which understands SAT as a very general phenomenon, while indeed “OBDD’s” is just one set of algorithms with small variations, solving just one task. The second requirement of [2], “an open, modular and extensible design that does not sacrifice efficiency” is a holy grail of software engineering, and generic and generative programming, fundamental for the **OKlibrary**, provides the next step.

UBCSAT ([9]) has a completely different focus, concentrating on one algorithm paradigm (local search), and this only within the context of boolean CNF; furthermore it is not a library of components, but offers the more convenient but less flexible approach of procedures which can be loaded within a given framework. As such UBCSAT is a useful tool and comes with the **OKlibrary**, yet as its main access point to the world of local search.\(^1\)

OpenSAT has merged into SAT4J, which again has a different focus: Rather practical solutions are sought (and nicely provided), and this only within the Java framework. For the beta-release the **OKlibrary** will also provide access to SAT4J, and forms of collaboration are envisaged to ease at least the flow of ideas.

\(^1\)As an example for the services offered by the **OKlibrary**, we provide automatic installation and a corrected version suitable now also for 64 bit platforms.
2 A hierarchy of 3 languages

The OKlibrary is in a sense a “multi-language library”, however it avoids “competing” languages for the same task, but every language is assigned a special purpose:

- For the “external sources”, any programming language is possible; in specialised modules linking to these packages happens (for example to interactive theorem provers like Coq, to constraint satisfaction solvers like Minion, or to a model checking system like Alloy), and there also programs in related programming languages are written, but this form of programming is considered as special-purpose programming only, similar to programming for the build-system (in the make-programming-language, or in the bash-shell-programming language).

- General purpose languages like Python (used by the computer algebra system Sage), Java (used by the SAT library SAT4J) or Ocaml (used by the SMT-solver DPT) on the other hand are to be avoided in the OKlibrary: If possible compilers and/or interpreters are provided in the external sources, but “balkanisation” of the library is not feasible (given the current state of software engineering), and C++ is the only general purpose programming language used in the OKlibrary.

- However, generic programming in C++ (or any other suitable language) has to face the enormous difficulties which come with the high level of abstraction (very many situations have to be imagined) together with the needed algorithmic efficiency (intricate knowledge about the special situations and their interaction needs to be used). To alleviate these (considerable) difficulties, development within the OKlibrary is envisaged to happen in three stages:

1. The first stage is given by writing “procedural specifications” in Lisp as extended by the Maxima computer algebra system. Here the design goal is to ignore (as completely as possible) efficiency concerns as well as abstraction, but to write small functions which directly communicate the underlying mathematical ideas.

2. The second stage is then given by the Aldor programming language, as extended by the computer algebra system Axiom: Here now we consider abstraction (abstract data types) in full generality, and also basic algorithmic question (about basic data structures) are taken into account, but all fine-grained algorithmic considerations are to be avoided, and especially everything concerned with the construction and destruction of objects (and resource management in general) is out of the picture.

3. Finally we have C++ which allows to express all concerns, and where especially we add to the previous levels concerns about resource management, about fine-grained algorithmic distinctions, and about the distinction between compile-time and run-time.

So we use it as an advantage that certain programming languages cannot express (appropriately) certain aspects of programming, however we do not make a fetish out of it, as if “thou shall not consider those details” — efficiency considerations are not “small details left to some employees” but are deeper properties which demand their own sphere but this best at later stages of the development, when a clearer picture emerged.
3 The general structure of the “OKplatform”

We now turn to an outline of the architecture of the OKlibrary.

3.1 A “holistic” library

The OKlibrary is a “world in itself” where all activities concerning the library happen inside the library, it can be reproduced and varied indefinitely, and every “clone” can show, localised and on its own, the full spectrum of possible behaviours. Especially all aspects of documentation, planning and discussion are integrated.

3.2 Generic programming

In recent years the paradigm of *generative programming* ([1]) has become a major player especially for the development of highly efficient and highly flexible libraries, exploring static polymorphism and code generators within the programming language. The main programming language supporting these features is C++. Due to the richer possibilities generative programming is especially suited to strengthen *generic programming* (“programming with type parameters”, a more powerful and more natural form of polymorphism than object-orientation), and the OKlibrary has the goal of developing dependent abstract data types (abstract data types depending on other abstract data types) for generalised SAT solving. To reach this stage, as discussed in Section 2 the “middle layer” of the Aldor/Axiom combination is an ideal guide, where Aldor provides the full power of dependent types while Axiom allows to specify semantic properties for abstract data types. The forthcoming new C++ standard (“C++ 09” due to the expected delivery in 2009) will add strong support for generic and generative programming to the core language; first prototypes will be available with gcc version 4.3.0.

3.3 The build system and the test system

Essential part of modern programming practice is continuous testing; to be effective, a system is needed which automatically detects all aspects of necessary re-compilations and re-running of tests (from the level of unit tests to the level of application tests). There exist several “unit testing frameworks”, but they all are too weak for the needs of a truly holistic library, where the tests are part of the services delivered by the library, and can be applied to newly developed components as well (which makes them “generic”; once the test systems becomes more complex we need also the ability to test the tests, and we arrive at the notion of a “higher order unit test” system). The management of compiling and running tests is one of the many responsibilities of the build system developed for the OKlibrary:

- Compilation of applications, (higher order) unit tests and applications tests.
- Execution of higher order unit tests and application tests, and making their results available.
- Enabling the expression of dependencies at various levels, including the global level, making the build system aware of all existing dependencies.
- Similar to the higher order unit test system, a system for complexity measurement. (This system has yet in the planning stage.)
• A configuration system for every aspect of the system.

• A documentation system at different levels:
  1. Top-level local and Internet index systems.
  2. Automatically extracted code annotations.
  3. Documentation pages for main aspects of modules and sub-modules.
  4. Demonstrations of applications.
  5. Access to the documentation for the external sources.

• Building external sources (and supporting their use).

• A planning system integrated into the system.

• Source control integrated into the system.

• Automated package building.

• Partially automated integration of services like the mailing-lists.

A central design goal of the OKlibrary, which is characteristic for “generative” or “active” libraries in the strong sense, is that configuration knowledge is (as complete as possible) expressed within the library; most of the time this means a high degree of automatisation (available within the library), and the build system is the backbone for these efforts.

3.4 External sources

Many tools, programs and libraries are relevant in the (wider) SAT context, and the OKlibrary wants to free its users from the considerable burden of installing, maintaining and updating these packages. Additionally we provide also corrections if necessary (and possible) and further documentation (at least how it fits into our environment). The current list of support “external sources” is as follows.

C/C++ libraries

- Boost (the C++ library of the highest standard, with sub-libraries ranging from diverse programming support over algorithms and data structures to mathematics)
- Mhash (collection of cryptological tools).

Compiler

- gcc (C++, C, Fortran, Java)
- Ocaml (functional programming language popular for logic applications)
- CLisp (the basis for Maxima and our “procedural specifications”).

Programming support Valgrind (a tool for detecting memory errors).

Buildsystem

- Git (distributed source-control with cryptographically secured history)
- Doxygen (a tool for extracting documentation from source code).
3 THE GENERAL STRUCTURE OF THE “OKPLATFORM”

Data management PostgreSQL (our database management system for experimental data).

Mathematics
- R (our main system for statistical data analysis)
- GMP (computation with arbitrary precision)
- Maxima (computer algebra system in Lisp)
- Sage (a computer algebra system embracing many others).

Logic Coq (proof assistant, which allows program extraction).

SAT March solvers, Minisat, UBCSAT.

Soon there will be more SAT solvers and SAT libraries (SAT4J and others; also SMT), but for the moment we just wanted to make sure that for each of the three currently dominant paradigms we have a relevant example. Soon also constraint solvers and libraries will be added. Regarding programming languages, Aldor will be added soon, regarding computer algebra Axiom, and regarding logic Isabelle.

3.5 Documentation

The anticipated library (research environment) is rather large, and good documentation is of importance. A “recursive” documentation system with functional differentiation has been designed to make sure that documentation can be added at every point of the tree in an easy and appropriate way (the build system will pick it up):

- Latex documentation is used for the mathematical underpinning and for the general ideas.
- The extracted documentation (html) provides more detailed information about all components.
- The docus-system (html) provides user-manuals for all aspects of the system.
- The demos-system (program code) provides easy examples on how to use the components provided by the library.

The planning process is integral part of the library, before anything is done is goes into the (again “recursive”) plans-structures, governed by a hierarchical milestones-system. This system takes especially account of the fact that in research environments development often has to be interrupted and taken up (sometimes years) later — the planning process guarantees that every activity can be interrupted at any time and continued (possibly by somebody else) at any time later. Compared to conventional models of development this is a rather postmodern (“fractured”) approach, which on the one hand makes the OKlibrary a permanent large construction place, but on the other hand ensures that development can react to the current needs, while maintaining a good quality of work.
3.6 Participation

Participation is planned to be organised in (roughly) three layers (exploiting the distributed source-control system):

1. At the core there is the group organised by me; I plan to devote substantial time into the library for the years to come, and I hope to mobilise also substantial funding for a group which also can handle over many years the more mundane tasks (and laborsome) aspects of the library development.

2. In the second layer we have the “external developers”, who according to their pleasure contribute to some modules, or add their own modules.

3. Finally, in the outer layers we have the clones of the OKlibrary which start their own universes, and which from time to time might return some acquisitions to the “mother ship”.

4 An overview on the parts of the OKlibrary

The OKlibrary is divided into “parts”, which each contain “supermodules”, which in turn (recursively) contain “modules”. To give an overview on the aims of the OKlibrary, we list in this section all parts of the OKlibrary, with short descriptions of their purpose. It must be said, that the various parts, supermodules and modules are at wildly different stages of development, often still in the planning process.

4.1 Applications

Here we have the main supermodules delivering components for applications of generalised SAT, at this time to colouring problems, cryptanalysis, Ramsey theory and Latin squares (which contains applications of SAT to Sudoku).

4.2 Combinatorics

SAT helps to solve combinatorial problems, and on the other hand is supported by a large variety of combinatorial theory. In this part we gather the main supermodules concerned with direct support of combinatorial structures and algorithms, either through interfaces to external sources (like for example the Boost graph library) or via direct approaches. The main modules at this time are for graphs and hypergraphs.

4.3 ComputerAlgebra

As discussed in Section 2, this part aims to implement all aspects of the generalised SAT world (and other parts of the OKlibrary) at radically simplified levels, the Lisp/Maxima level for “procedural specifications”, and the Aldor/Axiom level for “concept exploration”. At this time the main supermodules are on cryptology, graphs, hypergraphs, satisfiability and trees, where satisfiability yet contains modules on backtracking, branching tuples, clause-sets, constraint problems and propositional logic. One of the many services provided here shall be that we aim at providing all transformations belonging to the world of NP and related complexity
classes (and for many of them the efficiency reached at the Lisp/Maxima level will be sufficient).

Besides Maxima and Axiom also the computer algebra system Sage is provided, as an “umbrella” to give access especially to Gap (group theory — the study of symmetries) and computational commutative algebra (for example Singular — Gröbner bases and beyond).

4.4 DataStructures

Efficient algorithms exploit many sophisticated data structures. This part aims at delivering the (many) data structures which are not found in the C++ standard library or in Boost. Yet it is very preliminary, and only a module on trees exists.

4.5 Experimentation

In this part we provide tools for creating, running, evaluating and storing experiments, small and large-scale. We have a module implementing the evaluation mechanism of the SAT-2006-competition (as described in [8]; the idea is that such evaluation mechanisms can be used for comparing SAT-solvers by running a “mini-competition” according to well-established rules), a module with specification of SQL and XML databases for data about random problems (see [4]), a module with the OKgenerator (a strong generator for random problems; see [5, 3]), and first rudiments of the supermodule for the tools supporting running experiments (on diverse networks).

4.6 Logic

Obviously SAT has many relations to the world of logic in general; here we have first attempts at supermodules on finite model theory (for example providing a general specification mechanism via Fagin’s theorem and generalisations), first order logic (for example the various decidable classes are of interest here) and automated theorem proving (with obvious connections to SAT via the resolution calculus). We also find more theoretically oriented modules on bounded arithmetic (as a strong background theory of arithmetic), complexity theory (supplying an endless source of non-trivial algorithms) and computability (for educational purposes for example the original proof of the NP-completeness of SAT (resp. newer improvements) is to be implemented in the complexity-supermodule, and for this Turing machines are needed).

4.7 LogicalReflection

This part handles logical tools for construction and verification of algorithms and software. We consider verified components, automated program extraction, and especially the formalisation of upper bound proofs for the field of “exact algorithms” (with the goal of providing support for example for the larger and larger case-distinctions). Currently we are considering Coq and Isabelle.
4.8 Optimisation

Well-known application of optimisation to SAT exploit local search (in various forms) and linear programming, and also quadratic programming, semidefinite programming, convex and quasiconvex optimisation and genetic programming has been applied. General methods related to these theories will be provided in this part.

4.9 Programming

Here we find general (C++) programming tools which are needed somewhere in the library. The list includes support for container and sequence handling, components for error handling and input/output (in general), a supermodule delivering a powerful message-system (providing object-oriented messages with different languages and different levels, which can be checked at compile-time), tools for (template) metaprogramming and parallelism, parsing methods and handling of program options, and various utilities (like timestamps) and tools (for example for Latex-output). Also refactoring tools are approached.

As for the whole libraries, we try to (re)use as much given software as possible, and so many of the facilities here are kind of interfaces to external sources, however much needs to be provided directly, and parts of the library introduce novel concepts which need support on their own: A prime example is the higher-order test system (recall Subsection 3.3), and for example the supermodule on sequences contains generic tests for the various forms of iterators, so that if a model of some-iterator concept is implemented, then the general test facilities can be reused. Furthermore, external sources are often not well-documented, and then the OKlibrary aims at providing additional documentation (together with lots of demos).

4.10 Satisfiability

Naturally this part is the largest, containing all C++ components related to SAT solving together with high-level plans for satisfiability in other parts (like in the computer algebra system). The current list of supermodules is as follows:

1. Algorithms: generic satisfiability algorithms, contains currently modules AllSolutions, Autarkies, Backdoors, Backtracking, GraphDecomposition, Learning, LocalSearch, and SumProduct.
2. Assignments: handling partial and total assignments.
5. Heuristics: currently the theory of branching tuples (how to sensibly attach numbers to branchings measuring the quality), statistical analysis (evaluating progress) and structural recognition.
6. Interfaces: modules for input and output in various formats.
7. Optimisation: handles optimisation variants of SAT.
8. ProblemInstances: providing the fundamental building blocks for SAT solving, variables, “literals”, “clauses” and “clause-sets”, all concretely as well as in the abstract form as further discussed in Section 5. For example module ActiveClauseSets contains sub-modules Equivalences, LinearInequalities and PseudoBoolean, while ActiveClauses contains modules like InjectivityConstraints (similar to “AllDifferent”). The exploration of concepts (i.e., generalised interfaces) which allow for the high generality needed and allow high efficiency is under active development, and all components at this time are experimental prototypes (only).

9. ProofSystems: support for various proof systems like resolution, Frege systems or cutting planes.

10. Quantification: for QBF.

11. Reductions: yet (under construction) unit clause propagation ($r_1$), failed literal reduction ($r_2$) and, more generally, generalised unit clause propagation ($r_k$), with variations like local learning and/or combination with autarky search.

12. Solvers: here typically components are assembled together to yield “complete” solvers. Yet we have the OKsolver-family, containing plans for the new OKsolver’s, and the (improved) code for the old OKsolver (from the SAT-competition 2002), which is refurbished as a kind of “model solver” for a look-ahead solver using full failed literal reduction ($r_2$) at each node. Furthermore we have approaches at generic backtracking algorithms.

13. SpecialStructures: algorithms and data structures for special structures like hitting clause-sets and minimally unsatisfiable clause-sets.

14. Transformers: transforming other problem domains into SAT, for example crisp CSP and ladder logic, or SAT modulo theory; and generators. Much of such functionality is provided at the computer algebra level, while here we create components which need more sophistication (for example w.r.t. efficiency or preprocessing).

15. Variations: NAESAT, XSAT and others.

4.11 Statistics

Similar in spirit to the optimisation-part, here we provide general functionality regarding statistics, so for example support for the R system (which started as the open-source version of the S system, and is by far the largest such system), the main statistical system for the OKlibrary.

4.12 Structures

Here we find support for mathematical structures (in a general sense), like algebra, linear algebra, cryptology, number theory (for example “big numbers” via GMP), and sets (e.g., algorithms for sets and ordinal numbers).
4.13 System

Currently there are four supermodules in this part for the systemic aspects of the OKlibrary:

1. The build system (see Subsection 3.3).

2. The complexity system: yet under planning, and shall be an essential part of the OKlibrary, providing automatic performance measurement from the component-level to the application-level, together with database functionality to monitor the development.

3. The test system (see Subsection 3.3).

4. Finally we have a supermodule on legal issues, containing for example licence and copyright documentation and discussion, and tools for monitoring and installing licence standards in the OKlibrary.

4.14 Visualisation

Visualisation of for example graphs representing clause-sets has found quite some attraction in the SAT community. Of course, in the OKlibrary we provide support to use existing external sources, however often those facilities are not sufficient. So in this part we provide special tools for visualisation; at this time we have prototypes for visualising large (search) trees.

5 Generalised satisfiability problems

The key ideas for the notion(s) of generalised satisfiability problems can be summarised as follows:

1. “Generalised satisfiability problems” refine and rework the notions of “constraint” and “constraint satisfaction problem” through the lens of satisfiability algorithms (and thus SAT is considered in effect as an “abstract data type”). The difference to CSP is that generalised satisfiability problems are formulated in terms of partial assignments, where (at least implicitly) for CSP’s assignments have to be total for each constraint involved.

2. A key struggle is about differentiation between run-time and compile-time polymorphism, in order to make strong optimisations possible for the C++ compiler (especially regarding “C++ 2009”).

3. Resolution and the related conflict-learning process is obtained by the operation of partial assignments on the abstract domain of problem instances (in the sense of [7]). Thus “clauses” and “clause-sets” are always present, even if the problem instances do not speak about them.

4. A concept hierarchy of generalised satisfiability problems is build around the notion of “basic functionality” of a “constraint” in “answer” to a partial assignment:

   (a) An “active literal” is a satisfaction problem with “very fast” solution for all basic functionalities.
(b) The library supports also active literals in several variables as well as for example signed literals.

(c) “CNF-clauses” and “DNF-clauses” as sets of active literals.

(d) Conflict-learning is embodied by CNF-clauses, being the negation of DNF-clauses, considered as partial assignments.

(e) Thus partial assignments are closely related to (DNF-)clauses, but partial assignments offer different complexity guarantees than DNF-clauses.

(f) An “active clause” is a satisfaction problem with poly-time solution for most basic functionalities.

(g) An “active clause-set” is a general satisfaction problem, with no complexity guarantees on the basic functionalities, while using only one type of literal (the basic token for communication). To support heuristics, it offers an interface like an ordinary clause-set — “It looks like SAT, it behaves like SAT, but there might be no SAT in it”.

(h) Finally “alliances of active clause-sets” offer the possibility to employ different types of literals.

5. The library offers a complete duality between CNF and DNF (important for example for QBF).

6. Resolution and autarkies are considered as dual notions. Both resolution and autarkies are about the geometry of partial assignments in relation to the problem instance.

The library aims at offering as much as possible support for compile-time polymorphism(s). The vision is that algorithms are formulated for very general satisfiability problems, with non-boolean variables and very general forms of constraints, while the generative mechanism, when informed at compile time for example that the problem at hand just uses boolean variables and the input is an ordinary CNF, instantiates and simplifies all general constructions such that we obtain a program comparable in efficiency to one hand-written just for this special case.

References


REFERENCES


