A generic library for SAT problems

Higher-Order Unit Testing

Oliver Kullmann

University of Wales Swansea
Computer Science Department

ACCU Conference 2006

Introduction (SAT)

The subject of my talk:

The software-engineering aspects of the development of the `OKlibrary`, a generic (generative) (open source) C++ library for “(generalised) SAT solving”.

(This library could be understood as a (vast) generalisation of the Boost Graph library.)

Most people working in SAT doubt the possibility of a useful SAT library. We will see that they actually have reasons — it’s not clear whether we are (currently) ably to write a (useful) generic SAT library, but if, then I would consider it a breakthrough for the field.

Introduction (Testing)

The library shall deliver very general and (also) efficient components, with high flexibility on the functionality side **and** on the implementation side.

A strong verification process for the library will be crucial for its success. So testing cannot be left to the "back-office", but must be an integral part of the library.

After explaining the background, in this talk I'll focus on

generic higher order unit testing,

a framework which fully integrates testing and "normal programming".

("Normal programming" handles the input; "test programming" handles "programs" as input. We don't test implementations, but models of concepts.)

Overview

The role of SAT in this talk

In my talk I won’t dwell much on the SAT problem, but a basic understanding is needed in order to understand the basic design reasons.

The purpose of this section is:

- creating a feeling for the domain of the library
- en passant, to introduce a highly active and fascinating area of research in algorithms for “hard problems”.

I consider the (generalised) SAT problem as an ideal situation where generic programming should show its strength, combining generality *and* efficiency.

Examples of SAT problems

We consider **boolean expressions** like

$$F = (a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee c)$$

which for a programmer is the same as

```
bool F(bool a, bool b, bool c) {
  return (a or b) and (not a or c) and
      (not b or c);
}
```

The task is to find a **satisfying assignment**, that is, values for `a,b,c` such that $F(a, b, c) = 1$ holds (if no satisfying assignment exists, then F is unsatisfiable). This F has exactly 3 satisfying assignments: those assignments where c is 1, while at least one of a, b is 1.

Computations as boolean functions

Consider an idealised computer $\mathfrak{C}$, which takes n input bits

$$b_1, \ldots, b_n \in \{0, 1\}$$

and computes one output bit

$$\mathfrak{C}(b_1, \ldots, b_n) \in \{0, 1\}.$$

(The program is *hard-wired* into the computer, and solves a *decision problem*.)

A fundamental insight: Given $\mathfrak{C}$, we can efficiently construct a boolean formula $F(b_1, \ldots, b_n)$ computing the same function from $\{0, 1\}^n$ to $\{0, 1\}$ as $\mathfrak{C}$, where the size of F is not much bigger than the *number of steps* performed by $\mathfrak{C}$ in the worst case.

(Intuitively, we simulate the arbitrary computation by a simple digital computer, using and-, or-, and not-gates.)

NP completeness

We see (using suitable encodings): Solving a SAT problem F in n variables is “the same” as solving an equation

$$\mathfrak{C}(x_1, \dots, x_n) = y,$$

that is, finding some input $(x_1, \dots, x_n)$ which yields a certain prescribed output y, where the size of the SAT problem $F(\mathfrak{C})$ is “proportional” to the worst case running time of the computation $\mathfrak{C}$.

Considering the computation of $\mathfrak{C}(x_1, \dots, x_n)$ as a feasible verification of an alleged solution $(x_1, \dots, x_n)$ to some problem “specified” by $\mathfrak{C}$, the previous argument shows:

SAT is representative for the problem of *finding* a solution, where we (only) have efficient means for *checking* whether an alleged solution of the problem really is a solution.

Upside down

Traditionally, NP-completeness of SAT meant “infeasibility of SAT” (since likely we don’t have algorithms to solve such a (disguised) general problem).

With the advent of (much) improved algorithms and hardware, nowadays the universal character of SAT is used positively:

Problems are translated into SAT, and
SAT solvers are applied.

So SAT solvers are “universal problem solvers”: The user has not to come up with his own algorithms and implementations, but can use standardised SAT “technology”.

Main application areas for SAT

The prototypical application area for SAT is **verification** (model checking), especially hardware verification:

> Systems and properties are specified in some logical language (temporal logics, for example), and translated (using various methods) into SAT.

A closely related area is **Constraint Verification** (CSP). Actually, SAT became more successful than CSP in suitable areas, which many people attribute to better data structures and high-performance implementations.

There are annual **SAT competitions** (first 1992, “officially revived” 2002), which are highly important for the success of solvers, especially for the success of solvers in industry.

(However, typically solvers are rewritten in industry labs, and kept secret there.)

SAT solvers

Most SAT solvers (in the annual SAT competition around 40-50 solvers participate) are

1. written in C++
2. available as Open Source
3. written from scratch, using (often quite heavily) copy-and-modify.

There is a Java library (SAT4J), used by Java aficionados. Also a C++ library (Simo) exists, but seems to be used only within one research group.

In general, most people involved don't believe that any library here can be successful.

Isn’t’ there a problem ... ???

The basic algorithm for SAT solving is backtracking (“D(P)LL”). For n variables this can take up to 2^n steps.

Interesting problems involve at least a few thousand variables; in some extreme cases even millions of variables.
2^{5000} is quite big; $2^{1000000}$ is still bigger ?! A miracle ?!?!

The situation is *very complicated*: Sometimes it might take a minute, sometimes a year, sometimes 10^{100} years (and sometimes more, but so well).

A complicated landscape

SAT solvers are characterised by

- excessive variation
- excessive coupling
- obsession with efficiency.

This makes it quite challenging for libraries.

The socio-economic background

The purpose of this section is to give you some understanding of the special "ecosystem" we are coming from:

1. Starting from a single successful application, a much more ambitious project was set out.
2. Unlike traditional C++ libraries, a much higher degree of extensibility of the library is envisaged.
3. Open source, but not in the "usual sense".

The `OKsolver`

1. From 1997 - 2000 I developed the `OKsolver` (my first C program, motivated by the Böhm solver).
2. Summer 2001 I purchased Bjarne Stroustroup’s book on C++, and revamped the `OKsolver` by using concrete classes.
3. In the first “new” SAT competition 2002 `OKsolver` won 2 first and 2 third prizes (out of 9 categories).

Reasonable design (via (procedural) abstract data types), but `OKsolver` got unmaintainable due to `#ifdef`’s (roughly 5000 lines of code).

(As a theoretical computer scientist (and mathematician) I don’t work on the solver (or on any code) for maybe half a year, then I return — that’s really a mental effort with a complex program).

From “solver” to “library”

So an “active library” was needed:

- my algorithmic and conceptual ideas have evolved (towards “generalised SAT”, using complex methods from discrete mathematics and combinatorics);
- a robust and powerful process was needed, allowing modularisation + efficiency of generic implementation in an integrated build, test, compilation, documentation and versioning environment.

C++ is a reasonable language for this purpose (especially generic programming is very natural for writing algorithms).

I read “Modern C++ Design” and “Generative Programming”, which looked very interesting, also from an intellectual point of view.

(big mistake)

The `OKlibrary`

February 2004 a 3-years EPSRC-funded project started (120000 British pounds): “An algorithmic platform for efficient satisfiability based problem solving”; travel money, computers, one assistant for 2 years.

Aims of the `OKlibrary`:

- introducing fundamental new concepts and algorithms for generalised SAT solving;
- realising them using advanced C++ techniques.

(Many good generic libraries out there from a software engineering point of view, but not written by specialists in the fields, while specialists in the field typically write bad software. I want to create a “bleading edge” library regarding software quality and algorithms/concepts.)

No compromises regarding the core. (Version 1.0 was scheduled for Feb 2005; we should get it by Feb 2007.)

The goals for “holistic” libraries

To make my life easier, I decided to write a “holistic library”:

1. Nothing is outside of the library. All services connected with the library are available with every “local copy” of the library.
2. The library extends the language, and the user further extends the library.
3. The library checks itself.

A holistic library thus is its own (closed) universe.

It’s like a fixed point $f(x) = x$.

Not unlike meta-programming in C++ (as generative programming), the higher levels (outer hulls) must be reflected back to the base level.

The master-plan

The users; the “business plan”

At the beginning exactly one “stakeholder”, our (tiny) group (currently Matthew Henderson and me).

Some related research groups, and potentially some final-year students in the department, are the first external testers.

From the library `OKsolver_2.0` emerges, (hopefully) with good success at the SAT 2007 competition.

This shall create to initial impetus for a more widespread use among researchers and research laboratories. (I would consider 10 external entities seriously using the `OKlibrary` as quite a success.)

And it shall be the basis for a new grant application, asking for 2-3 research assistants over 3 years period.

Regarding open source

The library will be made available under some open source policy (concrete licence not yet decided).

But there is no participation framework planned for the next, say, 3 years:

1. The very notion of a "generalised SAT problem" lies at the heart of the library and is a main research result. So in a sense the library shall create a "new market" (a new way of thinking about the SAT problem and its applications).
2. The foundations (the buildsystem, the testsystem, the complexity system, the concepts) have to be established before any external participation.

Controlled participation

After the foundations have been set, we definitely hope that components will be contributed from external sources. (In the best of all worlds, on a larger scale.)

However, this will not happen as for example with the Boost library (via some Internet access):

External research groups first must realise some “profit” from their extensions of the library (papers, conference contributions), before they might want to add these extensions to the “official library”. (This special form of usage was one of the original motivations for the concept of a holistic library, a library which can be **locally extended**.)

I expect such an integration of external code often to be non-trivial, accomplished perhaps by some research visits.

An extended understanding of generic libraries

The purpose of this section is twofold:

- Discussing the notion of a generic library, and our extension to “higher order generic libraries”.
- Discussing some general problems related to generic libraries in our experience.

Concepts

“Concepts” are generalised interfaces, or, better, generalised *abstract data types* (ADT’s).

They describe a (mathematical) class of “models” by specifying

1. Syntax (how do we use it?)
2. Semantics (what is guaranteed to happen?)
3. Complexity (how much resources will it use?)

(The complexity requirements extend ADT’s. They are important, but problematic: It is essential for syntactical and semantical requirements, that they state *non-negative requirements*, while complexity requirements in the “standard form” are negative requirements, and thus contradict generic programming.)

Generic libraries

The common point of view of a generic library is, that it delivers

1. Concepts
2. Models.

The library authors should also have checked that the "models" provided are actually models.

According to the concept of a holistic library, these checks should be made available to the user — so that he can extend the library in a reliable way.

Thus a "higher order generic library" should also deliver

3. means for verifying "models",
4. instantiations of these checks with the given models,
5. a way of creating new instantiations (as well as new concepts and new checks),
6. and a way of executing all *necessary* checks.

Conceptual problems (and solutions)

- Don't know the concepts (yet): **Concept Migration**
- Efficient, complex, generic algorithms need many, many concepts: **Concept Parameterisation**
- Syntax checking: Boost Concept Checking Library (and C++0x !)
- Semantics checking: **HO Unit Tests** (more later)
- Complexity checking: *performance measurements* (similar to unit tests), managed with a database ...

Practical problems

- new algorithms (requiring new features from existing components)
- permanent modifications of implementations and functionality
- efficiency and modularisation are hard to combine (one has to be able to “grasp” into the implementations)
- copy-and-modify much easier than writing a library (much less abstraction needed)
- usage of generic libraries can be complicated (from time to time the whole library falls onto you)
- very hard to get qualified programmers.

Functions of the build system

The build system consists of a number of (quite carefully written) make files.

The main functionality is:

- building all external software (including gcc and Boost in all supported versions and combinations)
- building and running the tests
- running and administrating the performance tests
- compiling the documentation (html (Doxygen) and latex)
- creating distributions.

Let’s have a look ... [here the library files have been visited, and as some example we considered the Messages module]

The meaning of tests

A “test” is understood as an

approximation of a (complete) semantical concept check.

(We need this horizon of *proven correctness* in order to give (some) meaning to the “test functions”.)

We consider “test functions” just as ordinary “functions”. We want to have the “same good design” for the test functions as for the rest of the software.

Our aim is to obtain

reliable, reusable “test functions”.

(Some Java frameworks speak of “reusable test assets”, but I doubt that this can be achieved without a strong template mechanism.)

Nonstandard standard features of our testsystem

- extensible framework for **test levels** (default “basic”, “full”, “extensive”);
- **three output streams** (“error”, “messages”, “log”) with support for creating output and distributing it to various places (using the Boost Iostream Library; recall that our tests are typically deeply nested(!));
- due to the integration with the build system, **registration** of tests (i.e., instantiation of test functions with alleged (dependent) models) and execution of tests (only the necessary ones) requires minimal effort (without any external code generation);
- the test system can test itself.

Unit/regression/integration testing ??

We use **only (HO) unit testing**:

- every application must be a trivial application of components, thus no need for integration testing;
- if a bug was found, then the corresponding test function is strengthened to cover this bug (and more), and the testfunction testing this testfunction is informed about this bug, whence no need to distinguish regression testing.

Permanent improvement of the test functionality is essential:

We start with simple ad-hoc tests (as in ordinary unit testing), and every time we touch the model we move towards a more complete, systematic test.

(The idea of “cheap unit tests” seems fundamentally flawed to me.)

Only “abstract tests”

The notion of a “test function” expresses that we *never* test some concrete implementation, but, rather, a test function is something like

$$\mathrm{test}(X)$$

where the argument X has the type of a (dependent) concept.

Standard unit tests correspond to $\mathrm{test}()$ — recall “No global variables!” ?!

In traditional terms, we only do “black box testing”.

Our notion of tests requires, that not only our implementations must be **testable**, but also our concepts!

Input and output of tests ?!

Tests are not well-integrated into normal programming activities because of their peculiar input/output nature:

- the input is a kind of a program;
- either there is no output (“appears to be alright”), or we want to have a precise description of the error detected.

This “strangeness” (higher-order-ness) of tests makes it hard to do test-programming.

And last, but not least, there is the specification problem: What does a test really achieve??

What tests what??

If a test fails, what is broken:

- the tested component ?
- the test ?
- the test system ?
- the environment ??
- the universe ???

(Remark: We test the test system with the test system itself.)

The answer is obvious:

- first, we always impose a (reasonable) partial order on the tests, so that some tests are “more basic” than others;
- second, **It’s a feature!** Every test also tests everything.

What exactly do we test?

Push better than pull

We said a test function is written as $f(X)$ — what is X ? If we “test concepts”, then X should be a class.

The problem are the associated types: To create some test data, (too) often complicated `enable-if`’s are needed (to query the unknown associated types).

I started this way, but found this approach creates too many artificial problems.

Better is to allow dependent concepts, that is, X is not a class but a **class template** (in general).

So we “push” instead of “pull”.

Testing class templates

Assume that we have a concept $\mathfrak{C}$ and a dependent model

```
namespace Module {
  template <typename T>
  class M { ... };
}
```

We want to test here whether every `M<T>` is a model of $\mathfrak{C}$. So we write a test function

```
namespace Module { namespace test {
   template <template <class T> class C>
   class M_T : public OKlib::TestBase {
         ... };
}}
```

Some technical remarks

- Namespaces are very important to us, and they are considered being part of the name.
- The namespace structure coincides with the directory structure.
- Primary functionality is only implemented by classes and class templates, while free-standing functions and function templates are (important) “syntactic sugar”.
- Testability is achieved by atomisation of functionality.
- The most basic coding style principle: Trust the language.
- Only one language (C++).
- Don’t design where you have no knowledge — but improve the design every time you touch something.
- Concepts are first given implicitly.

Concepts vs. dependent concepts

So we test the *template* `M`. Isn't this kind of testing an implementation?

Not really: Finally we need a **dependent concept** $\mathfrak{C}(T)$.

But we might have other dependent models `M2<T>`, `M3<A,B>` ?

Then we'll improve the test system, and factor out the **generic tests** for $\mathfrak{C}$ itself:

Generic tests for $\mathfrak{C}$ are **axioms**, which take a sequence of objects and check some properties.

(Note that "axiomatic tests" do not create objects.)

Divide and substitute

Our test functions do exactly one of the following:

- In case concept $\mathfrak{C}$ asks for properties $P_1, \ldots, P_n$, then the test function calls the subtests for these properties (“base-concepts”).
- Otherwise we either choose “representative data” or “representative types” (or both) and substitute them (preferably “lazily”).

(Remark: Sometimes “diagonal substitution is needed” to ease creation of objects.)

At the leaves of the “test tree” constructed in this way we finally are able to decide the property, and in case of a failure the test system reports (at various levels of explicitness — using messages(!)) the path to the leaf in the test tree.

Summary

- SAT is a major technique to solve hard problems.
- The `OKlibrary` is designed to have a life-time of at least 10 years, enabling complex super-polynomial time algorithm research and making it practical.
- The framework to achieve these (not so trivial) goals is given by "holistic libraries", and especially "generic higher order unit testing".

- Outlook
  - We did a lot of (practical) research into efficient generic algorithms and data structures — and much more is needed.
  - Hope to see you in 2 years, reporting what the library has achieved!

For Further Reading I

- For the P vs. NP problem a general introduction is `http://www.claymath.org/millennium/P_vs_NP`.
- For the SAT competition see `http://www.satcompetition.org`.
- The main SAT discussion forum is `http://www.satlive.org`.
- The main SAT journal is `http://www.isa.ewi.tudelft.nl/Jsat/`.

Krzysztof Czarnecki and Ulrich W. Eisenecker.
Generative Programming.
Addison-Wesley, 2000.