Beginner's Guide to CPROVER

Martin Brain*

January 4, 2017

1 Background Information

First off; read the CPROVER manual. It describes how to get, build and use CBMC and SATABS. This document covers the internals of the system and how to get started on development.

1.1 Documentation

Apart from the (user-orientated) CPROVER manual and this document, most of the rest of the documentation is inline in the code as `doxygen` and some comments. A man page for CBMC, goto-cc and goto-instrument is contained in the `doc/` directory and gives some options for these tools. All of these could be improved and patches are very welcome. In some cases the algorithms used are described in the relevant papers.

1.2 Architecture

CPROVER is structured in a similar fashion to a compiler. It has language specific front-ends which perform limited syntactic analysis and then convert to an intermediate format. The intermediate format can be output to files (this is what `goto-cc` does) and are (informally) referred to as "goto binaries" or "goto programs". The back-end are tools process this format, either directly from the front-end or from it's saved output. These include a wide range of analysis and transformation tools (see Section 1.5).

1.3 Coding Standards

CPROVER is written in a fairly minimalist subset of C++; templates and meta-programming are avoided except where necessary. The standard library is used but in many cases there are alternatives provided in `util/` (see Section 2.3.1) which are preferred. Boost is not used.

*But most of the content is from Michael Tautschnig

Patches should be formatted so that code is indented with two space characters, not tab and wrapped to 75 or 72 columns. Headers for doxygen should be given (and preferably filled!) and the author will be the person who first created the file.

Identifiers should be lower case with underscores to separate words. Types (classes, structures and typedefs) names must[1] end with a `t`. Types that model types (i.e. C types in the program that is being interpreted) are named with `_typet`. For example `ui_message_handlert` rather than `UI_message_handlert` or `UIMessageHandler` and `union_typet`.

1.4 How to Contribute

Fixes, changes and enhancements to the CPROVER code base should be developed against the `trunk` version and submitted to Daniel as patches produced by `diff -Naur` or `svn diff`. Entire applications are best developed independently (`git svn` is a popular choice for tracking the main trunk but also having local development) until it is clear what their utility, future and maintenance is likely to be.

1.5 Other Useful Code

The CPROVER subversion archive contains a number of separate programs. Others are developed separately as patches or separate branches.Interfaces are have been and are continuing to stablise but older code may require work to compile and function correctly.

In the main archive:

`CBMC` A bounded model checking tool for C and C++. See Section 2.3.9.

`goto-cc` A drop-in, flag compatible replacement for GCC and other compilers that produces goto-programs rather than executable binaries. See Section 2.3.10.

`goto-instrument` A collection of functions for instrumenting and modifying goto-programs. See Section 2.3.11.

Model checkers and similar tools:

`SatABS` A CEGAR model checker using predicate abstraction. Is roughly 10,000 lines of code (on top of the CPROVER code base) and is developed in its own subversion archive. It uses an external model checker to find potentially feasible paths. Key limitations are related to code with pointers and there is scope for significant improvement.

`Scratch` Alistair Donaldson's k-induction based tool. The front-end is in the old project CVS and some of the functionality is in `goto-instrument`.

[1] There are a couple of exceptions, including the graph classes

`Wolverine` An implementation of Ken McMillan's IMPACT algorithm for sequential programs. In the old project CVS.

`C-Impact` An implementation of Ken McMillan's IMPACT algorithm for parallel programs. In the old project CVS.

`LoopFrog` A loop summarisation tool.

`???` Christoph's termination analyser.

Test case generation:

`cover` A basic test-input generation tool. In the old project CVS.

`FShell` A test-input generation tool that allows the user to specify the desired coverage using a custom language (which includes regular expressions over paths). It uses incremental SAT and is thus faster than the naïve "add assertions one at a time and use the counter-examples" approach. Is developed in its own subversion.

Alternative front-ends and input translators:

`Scoot` A System-C to C translator. Probably in the old project CVS.

`???` A Simulink to C translator. In the old project CVS.

`???` A Verilog front-end. In the old project CVS.

`???` A converter from Codewarrior project files to Makefiles. In the old project CVS.

Other tools:

`ai` Leo's hybrid abstract interpretation / CEGAR tool.

`DeltaCheck?` Ajitha's slicing tool, aimed at locating changes and differential verification. In the old project CVS.

There are tools based on the CPROVER framework from other research groups which are not listed here.

2 Source Walkthrough

This section walks through the code bases in a rough order of interest / comprehensibility to the new developer.

2.1 `doc`

At the moment just contains the CBMC man page.

2.2 regression/

The regression tests are currently being moved from CVS. The `regression/` directory contains all of those that have been moved. They are grouped into directories for each of the tools. Each of these contains a directory per test case, containing a C or C++ file that triggers the bug and a `.dsc` file that describes the tests, expected output and so on. There is a Perl script, `test.pl` that is used to invoke the tests as:

```
../test.pl -c PATH_TO_CBMC
```

The `--help` option gives instructions for use and the format of the description files.

2.3 src/

The source code is divided into a number of sub-directories, each containing the code for a different part of the system. In the top level files there are only a few files:

`config.inc` The user-editable configuration parameters for the build process. The main use of this file is setting the paths for the various external SAT solvers that are used. As such, anyone building from source will likely need to edit this.

`Makefile` The main systems Make file. Parallel builds are supported and encouraged; please don't break them!

`common` System specific magic required to get the system to build. This should only need to be edited if porting CBMC to a new platform / build environment.

`doxygen.cfg` The config file for doxygen.cfg

2.3.1 util/

`util/` contains the low-level data structures and manipulation functions that are used through-out the CPROVER code-base. For almost any low-level task, the code required is probably in `util/`. Key files include:

`irep.h` This contains the definition of `irept`, the basis of many of the data structures in the project. They should not be used directly; one of the derived classes should be used. For more information see Section 3.1.

`expr.h` The parent class for all of the expressions. Provides a number of generic functions, `exprt` can be used with these but when creating data, subclasses of `exprt` should be used.

`std_expr.h` Provides subclasses of `exprt` for common kinds of expression for example `plus_exprt`, `minus_exprt`, `dereference_exprt`. These are the intended interface for creating expressions.

`std_types.h` Provides subclasses of `typet` (a subclass of `irept`) to model C and C++ types. This is one of the preferred interfaces to `irept`. The front-ends handle type promotion and most coercision so the type system and checking goto-programs is simpler than C.

`dstring.h` The CPROVER string class. This enables sharing between strings which significantly reduces the amount of memory required and speeds comparison. `dstring` should not be used directly, `irep_idt` should be used instead, which (dependent on build options) is an alias for `dstring`.

`mp_arith.h` The wrapper class for multi-precision arithmetic within CPROVER. Also see `arith_tools.h`.

`ieee_float.h` The arbitrary precision float model used within CPROVER. Based on `mp_integer`s.

`context.h` A generic container for symbol table like constructs such as namespaces. Lookup gives type, location of declaration, name, 'pretty name', whether it is static or not.

`namespace.h` The preferred interface for the context class. The key function is `lookup` which converts a string (`irep_idt`) to a symbol which gives the scope of declaration, type and so on. This works for functions as well as variables.

2.3.2 `langapi/`

This contains the basic interfaces and support classes for programming language front ends. Developers only really need look at this if they are adding support for a new language. It's main users are the two (in trunk) language front-ends; `ansi-c/` and `cpp/`.

2.3.3 `ansi-c/`

Contains the front-end for ANSI C, plus a variety of common extensions. This parses the file, performs some basic sanity checks (this is one area in which the UI could be improved; patches most welcome) and then produces a goto-program (see below). The parser is a traditional Flex / Bison system.

`internal_addition.c` contains the implementation of various 'magic' functions that are that allow control of the analysis from the source code level. These include assertions, assumptions, atomic blocks, memory fences and rounding modes.

The `library/` subdirectory contains versions of some of the C standard header files that make use of the CPROVER built-in functions. This allows

CPROVER programs to be 'aware' of the functionality and model it correctly. Examples include `stdio.c`, `string.c`, `setjmp.c` and various threading interfaces.

2.3.4 `cpp/`

This directory contains the C++ front-end. It supports the subset of C++ commonly found in embedded and system applications. Consequentially it doesn't have full support for templates and many of the more advanced and obscure C++ features. The subset of the language that can be handled is being extended over time so bug reports of programs that cannot be parsed are useful.

The functionality is very similar to the ANSI C front end; parsing the code and converting to goto-programs. It makes use of code from `langapi` and `ansi-c`.

2.3.5 `goto-programs/`

Goto programs are the intermediate representation of the CPROVER tool chain. They are language independent and similar to many of the compiler intermediate languages. Section 3.2 describes the `goto_programt` and `goto_functionst` data structures in detail. However it useful to understand some of the basic concepts. Each function is a list of instructions, each of which has a type (one of 18 kinds of instruction), a code expression, a guard expression and potentially some targets for the next instruction. They are not natively in static single-assign (SSA) form. Transitions are nondeterministic (although in practise the guards on the transitions normally cover form a disjoint cover of all possibilities). Local variables have non-deterministic values if they are not initialised. Variables and data within the program is commonly one of three types (parameterised by width): `unsignedbv_typet`, `signedbv_typet` and `floatbv_typet`, see `util/std_types.h` for more information. Goto programs can be serialised in a binary (wrapped in ELF headers) format or in XML (see the various `_serialization` files).

The `cbmc` option `--show-goto-programs` is often a good starting point as it outputs goto-programs in a human readable form. However there are a few things to be aware of. Functions have an internal name (for example `c::f00`) and a 'pretty name' (for example `f00`) and which is used depends on whether it is internal or being presented to the user. The `main` method is the 'logical' main which is not necessarily the main method from the code. In the output `NONDET` is use to represent a nondeterministic assignment to a variable. Likewise `IF` as a beautified `GOTO` instruction where the guard expression is used as the condition. `RETURN` instructions may be dropped if they precede an `END_FUNCTION` instruction. The comment lines are generated from the `locationt` field of the `instructiont` structure.

`goto-programs/` is one of the few places in the CPROVER codebase that templates are used. The intention is to allow the general architecture of program and functions to be used for other formalisms. At the moment most of

the templates have a single instantiation; for example `goto_functionst` and `goto_function_templatet` and `goto_programt` and `goto_program_templatet`.

2.3.6 `goto-symex/`

This directory contains a symbolic evaluation system for goto-programs. This takes a goto-program and translates it to an equation system by traversing the program, branching and merging and unwinding loops as needed. Each reverse goto has a separate counter (the actual counting is handled by `cbmc`, see the `--unwind` and `--unwind-set` options). When a counter limit is reach, an assertion can be added to explicitly show when analysis is incomplete. The symbolic execution includes constant folding so loops that have a constant number of iterations will be handled completely (assuming the unwinding limit is sufficient).

The output of the symbolic execution is a system of equations; an object containing a list of `symex_target_elements`, each of which are equalities between `expr` expressions. See `symex_target_equation.h`. The output is in static, single assignment (SSA) form, which is *not* the case for goto-programs.

2.3.7 `pointer-analysis/`

To perform symbolic execution on programs with dereferencing of arbitrary pointers, some alias analysis is needed. `pointer-analysis` contains the three levels of analysis; flow and context insensitive, context sensitive and flow and context sensitive. The code needed is subtle and sophisticated and thus there may be bugs.

2.3.8 `solvers/`

The `solvers/` directory contains interfaces to a number of different decision procedures, roughly one per directory.

prop/ The basic and common functionality. The key file is `prop_conv.h` which defines `prop_convt`. This is the base class that is used to interface to the decision procedures. The key functions are `convert` which takes an `exprt` and converts it to the appropriate, solver specific, data structures and `dec_solve` (inherited from `decision_proceduret`) which invokes the actual decision procedures. Individual decision procedures (named `*_dect`) objects can be created but `prop_convt` is the preferred interface for code that uses them.

flattening/ A library that converts operations to bit-vectors, including calling the conversions in `floatbv` as necessary. Is implemented as a simple conversion (with caching) and then a post-processing function that adds extra constraints. This is not used by the SMT or CVC back-ends.

dplib/ Provides the `dplib_dect` object which used the decision procedure library from "Decision Procedures : An Algorithmic Point of View".

cvc/ Provides the `cvc_dect` type which interfaces to the old (pre SMTLib) input format for the CVC family of solvers. This format is still supported by depreciated in favour of SMTLib 2.

smt1/ Provides the `smt1_dect` type which converts the formulae to SMTLib version 1 and then invokes one of Boolector, CVC3, OpenSMT, Yices, MathSAT or Z3. Again, note that this format is depreciated.

smt2/ Provides the `smt2_dect` type which functions in a similar way to `smt1_dect`, calling Boolector, CVC3, MathSAT, Yices or Z3. Note that the interaction with the solver is batched and uses temporary files rather than using the interactive command supported by SMTLib 2. With the `--fpa` option, this output mode will not flatten the floating point arithmetic and instead output the proposed SMTLib floating point standard.

qbf/ Back-ends for a variety of QBF solvers. Appears to be no longer used or maintained.

sat/ Back-ends for a variety of SAT solvers and DIMACS output.

2.3.9 `cbmc/`

This contains the first full application. CBMC is a bounded model checker that uses the front ends (`ansi-c`, `cpp`, goto-program or others) to create a goto-program, `goto-symex` to unwind the loops the given number of times and to produce and equation system and finally `solvers` to find a counter-example (technically, `goto-symex` is then used to construct the counter-example trace).

2.3.10 `goto-cc/`

`goto-cc` is a compiler replacement that just performs the first step of the process; converting C or C++ programs to goto-binaries. It is intended to be dropped in to an existing build procedure in place of the compiler, thus it emulates flags that would affect the semantics of the code produced. Which set of flags are emulated depends on the naming of the `goto-cc/` binary. If it is called `goto-cc` then it emulates GCC flags, `goto-armcc` emulates the ARM compiler, `goto-cl` emulates VCC and `goto-cw` emulates the Code Warrior compiler. The output of this tool can then be used with `cbmc` or `goto-instrument`.

2.3.11 `goto-instrument/`

The `goto-instrument/` directory contains a number of tools, one per file, that are built into the `goto-instrument` program. All of them take in a goto-program (produced by `goto-cc`) and either modify it or perform some analysis. Examples include `nondet_static.cpp` which initialises static variables to a non-deterministic value, `nondet_volatile.cpp` which assigns a non-deterministic value to any volatile variable before it is read and `weak_memory.h` which performs the necessary transformations to reason about weak memory models.

The exception to the "one file for each piece of functionality" rule are the program instrumentation options (mostly those given as "Safety checks" in the `goto-instrument` help text) which are included in the `goto-program/` directory. An example of this is `goto-program/stack_depth.h` and the general rule seems to be that transformations and instrumentation that `cbmc` uses should be in `goto-program/`, others should be in `goto-instrument`.

`goto-instrument` is a very good template for new analysis tools. New developers are advised to copy the directory, remove all files apart from `main.*`, `parseoptions.*` and the `Makefile` and use these as the skeleton of their application. The `doit()` method in `parseoptions.cpp` is the preferred location for the top level control for the program.

2.3.12 `linking/`

Probably the code to emulate a linker. This allows multiple 'object files' (goto-programs) to be linked into one 'executable' (another goto-program), thus allowing existing build systems to be used to build complete goto-program binaries.

2.3.13 `big-int/`

CPROVER is distributed with its own multi-precision arithmetic library; mainly for historical and portability reasons. The library is externally developed and thus `big-int` contains the source as it is distributed. This should not be used directly, see `util/mp_arith.h` for the CPROVER interface.

2.3.14 `xmllang/`

CPROVER has optional XML output for results and there is an XML format for goto-programs. It is used to interface to various IDEs. The `xmllang/` directory contains the parser and helper functions for handling this format.

2.3.15 `floatbv/`

This library contains the code that is used to convert floating point variables (`floatbv`) to bit vectors (`bv`). This is referred to as 'bit-blasting' and is called in the `solver` code during conversion to SAT or SMT. It also contains the abstraction code described in the FMCAD09 paper.

3 Data Structures

This section discusses some of the key data-structures used in the CPROVER codebase.

3.1 irept

There are a large number of kind of tree structured or tree-like data in CPROVER. `irept` provides a single, unified representation for all of these, allowing structure sharing and reference counting of data. As such `irept` is the basic unit of data in CPROVER. Each `irept` contains[2] a basic unit of data (of type `dt`) which contains four things:

`data` A string[3], which is returned when the `id()` function is used.

`named_sub` A map from `irep_namet` (a string) to an `irept`. This is used for named children, i.e. subexpressions, parameters, etc.

`comments` Another map from `irep_namet` to `irept` which is used for annotations and other 'non-semantic' information

`sub` A vector of `irept` which is used to store ordered but unnamed children.

The `irept::pretty` function outputs the contents of an `irept` directly and can be used to understand an debug problems with `irept`s.

On their own `irept`s do not "mean" anything; they are effectively generic tree nodes. Their interpretation depends on the contents of result of the `id` function (the `data`) field. `util/irep_ids.txt` contains the complete list of `id` values. During the build process it is used to generate `util/irep_ids.h` which gives constants for each id (named `ID_*`). These can then be used to identify what kind of data `irept` stores and thus what can be done with it.

To simplify this process, there are a variety of classes that inherit from `irept`, roughly corresponding to the ids listed (i.e. `ID_or` (the string `"or''`) corresponds to the class `or_exprt`). These give semantically relevant accessor functions for the data; effectively different APIs for the same underlying data structure. None of these classes add fields (only methods) and so static casting can be used. The inheritance graph of the subclasses of `irept` is a useful starting point for working out how to manipulate data.

There are three main groups of classes (or APIs); those derived from `typet`, `codet` and `exprt` respectively. Although all of these inherit from `irept`, these are the most abstract level that code should handle data. If code is manipulating plain `irept`s then something is wrong with the architecture of the code.

Many of the key descendent of `exprt` are declared in `std_expr.h`. All expressions have a named subfield / annotation which gives the type of the expression (slightly simplified from C/C++ as `unsignedbv_typet`, `signedbv_typet`, `floatbv_typet`, etc.). All type conversions are explicit with an expression with `id() == ID_typecast` and an 'interface class' named `typecast_exprt`. One key descendent of `exprt` is `symbol_exprt` which creates `irept` instances with

[2]Or references, if reference counted data sharing is enabled. It is enabled by default; see the `SHARING` macro.

[3]When `USE_DSTRING` is enabled (it is by default), this is actually a `dstring` and thus an integer which is a reference into a string table

the id of "symbol". These are used to represent variables; the name of which can be found using the `get_identifier` accessor function.

`codet` inherits from `exprt` and is defined in `std_code.h`. They represent executable code; statements in C rather than expressions. In the front-end there are versions of these that hold whole code blocks, but in goto-programs these have been flattened so that each `irept` represents one sequence point (almost one line of code / one semi-colon). The most common descendents of `codet` are `code_assignt` so a common pattern is to cast the `codet` to an assignment and then recurse on the expression on either side.

3.2 goto-programs

The common starting point for working with goto-programs is the `read_goto_binary` function which populates an object of `goto_functionst` type. This is defined in `goto_functions.h` and is an instantiation of the template `goto_functions_templatet` which is contained in `goto_functions_template.h`. They are wrappers around a map from strings to `goto_programt`'s and iteration macros are provided. Note that `goto_function_templatet` (no `s`) is defined in the same header as `goto_functions_templatet` and is gives the C type for the function and Boolean which indicates whether the body is available (before linking this might not always be true). Also note the slightly counter-intuitive naming; `goto_functionst` instances are the top level structure representing the program and contain `goto_programt` instances which represent the individual functions. At the time of writing `goto_functionst` is the only instantiation of the template `goto_functions_templatet` but other could be produced if a different data-structures / kinds of models were needed for functions.

`goto_programt` is also an instantiation of a template. In a similar fashion it is `goto_program_templatet` and allows the types of the guard and expression used in instructions to be parameterised. Again, this is currently the only use of the template. As such there are only really helper functions in `goto_program.h` and thus `goto_program_template.h` is probably the key file that describes the representation of (C) functions in the goto-program format. It is reasonably stable and reasonably documented and thus is a good place to start looking at the code.

An instance of `goto_program_templatet` is effectively a list of instructions (and inner template called `instructiont`). It is important to use the copy and insertion functions that are provided as iterators are used to link instructions to their predecessors and targets and careless manipulation of the list could break these. Likewise there are helper macros for iterating over the instructions in an instance of `goto_program_templatet` and the use of these is good style and strongly encouraged.

Individual instructions are instances of type `instructiont`. They represent one step in the function. Each has a type, an instance of `goto_program_instruction_typet` which denotes what kind of instruction it is. They can be computational (such as `ASSIGN` or `FUNCTION_CALL`), logical (such as `ASSUME` and `ASSERT`) or informational (such as `LOCATION` and `DEAD`). At the time of writing there are 18

possible values for `goto_program_instruction_typet` / kinds of instruction. Instructions also have a guard field (the condition under which it is executed) and a code field (what the instruction does). These may be empty depending on the kind of instruction. In the default instantiations these are of type `exprt` and `codet` respectively and thus covered by the previous discussion of `irept` and its descendents. The next instructions (remembering that transitions are guarded by non-deterministic) are given by the list `targets` (with the corresponding list of labels `labels`) and the corresponding set of previous instructions is get by `incoming_edges`. Finally `instructiont` have informational `function` and `location` fields that indicate where they are in the code.