HACKT

Hierarchical Asynchronous Circuit Kompiler Toolkit

David Fang

This manual describes the usage and operation of HACKT's tools.

This document can also be found online at http://www.csl.cornell.edu/~fang/hackt/hackt.

The main project home page is http://www.csl.cornell.edu/~fang/hackt/.

Short Contents

Table of Contents

1 Introduction

This document is a usage guide for the set of HACKT executables.

`hackt` is a command to dispatch one of many programs in its collection of tools. The general usage is:

`hackt [`*general options*`] `*command*` [`*command arguments*`]`

For example, '`hackt version`' prints the version and configuration information for the tools. Currently, there are no general options to `hackt`, but some may be added in the future.

Commands will give a brief summary of their options when invoked without command arguments, or when passed '`-h`' for help. Some common subprograms also have equivalent single-name commands that are installed by `make install`.

The following topics are *not* covered in this guide:

- language – covered in '`hac.pdf`', built in '`dox/lang`', installed in '*prefix*`/share/hackt/doc/pdf/`'.
- simulators – covered in separate guides '`hacprsim.pdf`' and '`hacchpsim.pdf`'.

All documents come in the following formats: '`.pdf`', '`.ps`', '`.html`', '`.info`', installed in '*prefix*`/share/hackt/doc/`'.

2 Compiler

TODO: figure of compile flow and phases.

2.1 Compile

The first compile phase produces a parsed and partially checked object file given an input text (source) file.

`haco` [*options*] *source object* [Program]
Compile HAC source to object file.

The source file is a text file in the HAC language. The object file, if given, is the result of the compile. If the object file is omitted, the program just reports the result of complation without producing an object file.

Options:

`-h` [User Option]
Show usage.

`-I` *path* [User Option]
Adds include path *path* for importing other source files (repeatable).

`-d` [User Option]
Produces text dump of compiled module, like `hacobjdump` in Section 4.2 [Objdump], page 9.

`-f` *optname* [User Option]
general compile flags (repeatable) where *optname* is one of the following:

- '`dump-include-paths`': dumps '`-I`' include paths as they are processed
- '`dump-object-header`': (diagnostic) dumps persistent object header before saving
- '`no-dump-include-paths`': suppress feedback of '`-I`' include paths
- '`no-dump-object-header`': suppress persistent object header dump

Dialect flags (for ACT-compatbility):

- '`export-all`': Treat all definitions as exported, i.e. no export checking.
- '`export-strict`': Check that definitions are exported for use outside their respective home namespaces (default, ACT).
- '`namespace-instances`' Allow instance management outside global namespace (default). Negatable with `no-` prefixed. ACT mode: '`no-namespace-instances`'.
- '`array-internal-nodes`' Allow implicit arrays of internal nodes in PRS (default). Negatable with `no-` prefixed. ACT mode: '`no-array-internal-nodes`'.

'`ACT`' is a preset that activates all ACT-mode flags for compatibility.

`-M` *depfile* [User Option]
Emit import dependencies in file *depfile* as a side-effect. Useful for automatic dynamic dependency-tracking in Makefiles.

`-o` *objfile* [User Option]
: Names *objfile* as the output object file to save. This is an alternative to naming the object file as the second non-option argument.

`-p` [User Option]
: Expect input to be piped from stdin rather than a named file. Since the name of the input file is omitted in this case, the only non-option argument (if any) is interpreted as the name of the output object file.

`haco` is provided as a single-command alias to `hackt compile`[1].

TODO: quick explanation of parse errors.

2.2 Preprocessor

`hacpp` is a preprocessor that expands imports, much like `cpp` expands `#include` and other preprocessor directives. This can be convenient for flattening hierarchies of imported sources into a self-contained file.

One nice feature is that the output (by default) preserves precise information about which files were imported, so compiling a flattened source file should result in the same error messages as compiling the original source file.

2.3 Create

The create phase generates footprints for each complete type once, so that instances of the same type may share the same footprint. Each type's definition is sequentially unrolled and expanded (instantiations and connections) recursively after substituting meta-parameter arguments. The input object file is the result of `haco`.

`haccreate` [*options*] *in-object out-object* [Program]
: Further compiles an object file through the create phase.

`haccreate` is provided as a single-command alias to `hackt create`.

All programs that normally expect object files as inputs can also invoke the compiler on a source file with the following options.

`-c` [User Option]
: Indicate that input file is a source, not object file, and needs to be compiled.

`-C` *opts* [User Option]
: When compiling source, forward options *opts* to the compiler-driver. **Suggestion**: when passing compiler-options on the command-line, wrap in "double-quotes" to group a list of arguments into a single string in the shell.

[1] Thus named because we use `.haco` as the extension for compiled object files

2.4 Allocate

The alloc phase expands the entire instance hierarchy top-down, so that every unique instance has its own state in memory. The input object file is the result of `haccreate`, but will automatically run `create` on the object file if necessary.

`hacalloc` *in-object out-object* [Program]
Further compiles an object file through the alloc phase.

`hacalloc` is provided as a single-command alias to `hackt alloc`. `hacalloc` supports the same options as `haccreate`.

In summary, the create, and allocate compile phases automatically run the necessary prerequisite phases where needed. All developed tools should also implement this behavior.

2.5 Instance Dump

We provide a utility to print instance and type information, which can be used by other programs for performing text-based queries. `hacinstdump` takes an object file and prints a table of all instances in the (flattened) hierarchy. The table contains information about the type of each named instance. The program takes an object file as an argument, and has no other options.

2.6 Conventions

As a convention, we name our object files according to the last phase with which it was processed or produced. The tools are actually extension agnostic; you can use whatever file extensions you like for both source and object files.

- `.haco` – compiled only
- `.hacf` – preprocessed source only
- `.haco-c` – compiled and created
- `.haco-a` – compiled and allocated

These suffices can be used to define `make` rules. Examples of Makefile templates can be found in the distribution `lib/mk` or installed in *`prefix`*`/share/hackt/mk/hackt.mk`.

2.7 Examples

In this section, we use the following source '`inv.hac`' as our input example.

```
defproc inv(bool a, b) {
prs {
  a    -> b-
  ~a   -> b+
}
}

bool x, y;
inv Z(x, y);
```

This defines an inverter process `inv` with public boolean ports `a` and `b`. The top-level declares boolean nodes `x` and `y`, which are connected to the ports of instantiated inverter `Z`.

A more comprehensive description of the language can be found built in '`dox/lang/hac.pdf`' or installed as '`/install/share/hackt/doc/pdf/hac.pdf`'.

TODO: fill in uses of example

3 Shell Interpreter

The `hackt shell` is intended as a general purpose tool for manipulating object files, directing fine-grain control over partial compilation, mechanical program transformations and analyses, etc. This project is barely begun. It is merely an object of curiosity at the moment. It currently supports no commands and is, thus, utterly useless. The only feature of the shell is the ability to escape to the parent shell and run commands. For example,

```
hacksh> !date
Fri May 26 18:20:47 EDT 2006
```

Not very exciting yet.

4 Diagnostics

This chapter describes some diagnostic commands of `hackt` tools.

4.1 Version

The `version` command just tells you the configuration with which `hackt` was compiled and installed. The output may looks something like the following:

```
$ hackt version
Version: hackt 0.1.4-devel-20060508
CVS Tag: HACKT-00-01-04-main-00-79-03-CHP-02-01
Configured with: '--enable-fun' '--with-editline=/usr'
'YACC=/usr/bin/yacc' '--prefix=/Users/davidfang/local' '-C'
'CC=ccache gcc' 'CXX=ccache g++'
build-triplet: powerpc-apple-darwin7.9.0
c++: g++ (GCC) 3.3 20030304 (Apple Computer, Inc. build 1640)
AM_CPPFLAGS: -I../../../src -I/usr/include
AM_CXXFLAGS: -pipe -ansi -pedantic-errors -Wold-style-cast
-Woverloaded-virtual -W -Wall -Wundef -Wshadow
-Wno-unused-parameter -Wpointer-arith -Wcast-qual -Wcast-align
-Wconversion -Werror
AM_LDFLAGS: -L/usr/lib
config-CXXFLAGS: -g -O2
config-CPPFLAGS:
config-LDFLAGS:
config-LIBS: -ledit  -lncurses
lex: flex version 2.5.4
yacc: /usr/bin/yacc
readline: BSD EditLine (histedit interface) ver. 2.9
build-date: Wed May 10 18:24:59 EDT 2006
```

This information is especially useful for reporting bugs. A list of known successful configurations is in the top source directory's `BUILDS` file. Reports of new configurations are always welcome.

4.2 Object Dump

`hackt` also provides `objdump` as a command for viewing the contents of a compiled object file as (questionably) human-readable text. The regression test suite uses `objdump` heavily to verify the contents of object files as they are transformed through the various compile phases. Occasionally, it may be useful to the casual or curious user for bug tracking.

`hacobjdump` *object-file* [Program]

Prints textual dump of compiled object file *object-file*. Object file may be compiled to any phase.

5 Legacy Compatibility

NOTE: this section is somewhat of redundant with the `cast2hac` directory documentation. Please refer to `cast2hac.pdf` for a guide on migrating to the new `hackt` tools.

This section is only useful to those who have used the legacy CAST tools. We provide some tool commands for use with legacy CAST tools. The aim is to provide a bridge from old tools to `hackt`.

5.1 CAST Flatten

The old CAST tool chain uses flattened text files as input to other tools. We provide similar functionality with HACKT's `cflat` command, which is also installed under the alias `hflat`.

`hflat` *mode* [*options*] *in-object* [Program]

Emulate the behavior of legacy `cflat`. Modes and options are explained below.

Instead of reading in the source file directly, it reads a compiled object file. (Later, we may add an option to read a source file directly.) If the object file is not already in the allocated state (Section 2.4 [Allocate], page 5), then it will automatically invoke the allocation phase before doing its real work. The options and modes are described in Section 5.1.1 [CFLAT Options], page 11.

Starting with our example from Section 2.7 [Program Examples], page 5. we compile `inv.haco` first.

```
$ hackt compile inv.hac inv.haco
```

We then produce flattened text output with the command:

```
$ hackt cflat prsim inv.haco
```

which results the following output, suitable for legacy `prsim`:

```
"x" -> "y"-
~"x" -> "y"+
= "x" "Z.a"
= "y" "Z.b"
```

This can be piped directly into `prsim` or saved to a file for later use.

5.1.1 CFLAT Options

General options:

`-c` [User Option]

Indicate that input file is source, as opposed to an object file, and needs to be compiled.

`-C` *opts* [User Option]

When compiling input source, forward options *opt* to the compiler driver.

`hflat` provides convenient and fine-grain control over the output text format. Options can be divided into two categories, *modes* and *flags*. Flags control individual traits of the output format, whereas modes are presets of traits, named after specific tools. The presets are set to emulate the formats expected by the legacy tools as closely as possible. Currently, the following list of modes is supported:

`prsim` [`cflat` option]
: `prsim` output mode.

`lvs` [`cflat` option]
`LVS` [`cflat` option]
`java-lvs` [`cflat` option]
: LVS output mode. The java-lvs option is a slight variant from the traditional lvs.

`ergen` [`cflat` option]
: `ergen` output mode.

`alint` [`cflat` option]
: `alint` output mode.

`prlint` [`cflat` option]
: `prlint` output mode.

`prs2tau` [`cflat` option]
: `prs2tau` output mode.

`connect` [`cflat` option]
: `connect` output mode.

`check` [`cflat` option]
: `check` output mode.

`wire` [`cflat` option]
: `wire` output mode.

`aspice` [`cflat` option]
`Aspice` [`cflat` option]
: `aspice` output mode.

`ADspice` [`cflat` option]
: `ADspice` output mode.

`default` [`cflat` option]
: `default` output mode.

TODO: make table summarizing the flags implied by each preset mode.

Other non-preset options can be used to fine-tune and customize the output format. All options except the '`connect-*`' options may also be prefixed with '`no-`' for negation, e.g. '`-f no-sizes`' disables printing of sized production rule literals. The following '`-f`' flags are supported:

`no-connect` [`cflat -f` option]
`connect-none` [`cflat -f` option]
: Suppress printing of aliases.

`connect-equal` [`cflat -f` option]
: Print aliases with style: '`= x y`'.

`connect-connect` [`cflat -f` option]
: Print aliases with style: '`connect x y`'.

`connect-wire` [`cflat -f` option]
: Print aliases with style: '`wire x y`'.

`include-prs` [`cflat -f` option]
`exclude-prs` [`cflat -f` option]
`no-include-prs` [`cflat -f` option]
`no-exclude-prs` [`cflat -f` option]
: Include or exclude production rules from output.

`precharges` [`cflat -f` option]
`no-precharges` [`cflat -f` option]
: Print or hide precharge expressions.

`self-aliases` [`cflat -f` option]
`no-self-aliases` [`cflat -f` option]
: Includes or exclude aliases '`x = x`'.

`quote-names` [`cflat -f` option]
`no-quote-names` [`cflat -f` option]
: Wrap all node names in "quotes".

`node-attributes` [`cflat -f` option]
`no-node-attributes` [`cflat -f` option]
: Whether or not to print node attributes.

`split-instance-attributes` [`cflat -f` option]
`join-instance-attributes` [`cflat -f` option]
: Determines whether to print instance attributes (including nodes) on a single line like:

```
@ "node" attr1 attr2 attr3 ...
```

or one attribute per line:

```
@ "node" attr1
@ "node" attr2
@ "node" attr3
...
```

`SEU` [`cflat -f` option]
`no-SEU` [`cflat -f` option]
: Enable single-event-upset mode for selected tool.

`check-mode` [`cflat -f` option]
`no-check-mode` [`cflat -f` option]
: Silences `cflat` output while traversing hierarchy. Useful only as a diagnostic tool for debugging.

`wire-mode` [`cflat -f` option]
`no-wire-mode` [`cflat -f` option]
: Accumulate aliases in the form: '`wire (x,y,...)`'

`dsim-prs` [`cflat -f` option]
`no-dsim-prs` [`cflat -f` option]
: Wraps prs in: '`dsim { ... }`'

`sizes` [`cflat -f` option]
`no-sizes` [`cflat -f` option]
: Prints rule literals with `<size>` specifications.

Preset modes are just combinations of the individual mode modifiers.

Command Index

Variable Index

Concept Index