CAST to HAC Migration Guide

David Fang

This document describes language and tool migration from old CAST to HAC tools.
This document can also be found online at http://www.csl.cornell.edu/~fang/hackt/cast2hac.
The main project home page is http://www.csl.cornell.edu/~fang/hackt/.

Published by ...

Short Contents

Table of Contents

1 Introduction

The purpose of this document is to assist CAST users in migrating old sources to the HAC language for use with the HACKT tools. This document assumes the reader is already familiar with the CAST language and some of its tools. This guide does not cover the new features of the HAC language; those are covered in the language reference.

2 Types

2.1 Fundamental Types

The names of the primitive types have changed. Types that correspond to compile-time meta parameters are prefixed with 'p', such as `pint` and `pbool`.

Here is a summary of changes in fundmental types from CAST (first column) to HAC (second column).

CAST	**HAC**
`node`	`bool`
N/A	`int`
`int`	`pint`
`bool`	`pbool`
N/A	`preal`

The new types `preal` and `int` are explained in the HAC language reference.

2.2 User-defined Types

For the purposes of migrating from CAST, we restrict our attention in the rest of this document to *processes*, the only kind of user-defined metaclass in CAST. HAC supports data and channel metaclasses, which have no equivalent in CAST. Chapter 3 [Definitions], page 5, describes the changes in defining processes.

3 Definitions

Definitions use the keywords `defproc`, `defchan`, and `defdata`. The old CAST keyword `define` can simply be replaced with `defproc`.

TODO: write more on newer port restrictions...

3.1 Ports

In CAST, port declarations were allowed to be sparse, whereas in HAC, only dense declarations are allowed.

The following CAST definitions were legal:

```
define foo()(node x[1..3]) { }
define bar()(node x[0..2]) { }
```

However, in HAC, there is no equivalent to declaring port instances that start with non-zero indices. Note, that in CAST, the first set of parentheses in each definition are reserved for template parameterization. Thus, the above definition of `foo` cannot be expressed in HAC, whereas the definition of `bar` could be rewritten:

```
defproc bar(bool x[3]) { }
```

For more on dense and sparse arrays and their declarations, See Chapter 5 [Arrays], page 9. Chapter 9 [Templates], page 17 describes how to generalize definitions using template parameters.

4 Instances

CAST allowed declarations of instances to be anonymous, which would result in automatically generated names such as `_a_0`. HAC, however, requires all instances to be named. Anonymous instances were sometimes used to declare arrays of a particular type without namimng the instances. (Example?)

5 Arrays

In CAST, the dimensions were declared between the type-identifier and the instance-identifier. Arrays in HAC are syntactically C-style, where the dimensions of the array follow the array's identifier.

Sparse arrays are also supported but with different syntax. In HAC, sparse arrays are *always* declared using a range expression in the indices. e.g. '`inv x[1..1];`' declares a sparse 1D array populated at index 1. (Yes, some of you may find this inconvenient.)

The following table shows some examples of equivalent declarations in CAST and HAC, where `inv` is defined as a type.

Examples of dense and sparse array declarations:

CAST	**HAC**	**meaning**
`inv[2] x;`	`inv x[2];`	1D array with indices 0..1
`inv[2] x, y;`	`inv x[2], y[2];`	2 1D arrays with indices 0..1
`inv[2][3] x;`	`inv x[2][3];`	2D array with indices 0..1,0..2
`inv x[2];`	`inv x[2..2];`	sparse 1D array indexed 2 only
`inv x[2..4];`	`inv x[2..4];`	sparse 1D array indexed 2..4
`inv x[2], x[4];`	`inv x[2..2], x[4..4];`	sparse 1D array indexed 2, 4 only

A common pitfall is to declare sparse index of an array and pass port connections in the same statement, such as: '`inv w[2](x, y, z);`'. This illegal statement tries to declare an *array* indexed 0..1, and connect both instances with the same port parameters. In HAC, one cannot declare a collection and connect its ports in the same statement as in CAST, however, one may declare a scalar instance and connect its ports in the same statement. The proper way to instantiate and connect a sparse instance is to use a sparse range, just like in a sparse declaration: '`inv w[2..2](x, y, z);`'.

6 Connections

6.1 Scalar instance connections

CAST allowed a definition to be connected with fewer port arguments than port parameters, assuming that the trailing ports remain unconnected externally. HAC forbids this error-prone mismatch, and requires one to pass in blanks in place of all unconnected ports, even trailing ports.

For example, given a `defproc` (`define`) `inv` with three `bool` (`node`) ports and external nodes `x, y, z`, The following table shows what declaration connections are legal:

declaration	**CAST**	**HAC**
`inv x;`	valid	valid
`inv x();`	valid	invalid
`inv x(x, y, z);`	valid	valid
`inv x(x, y);`	valid	invalid
`inv x(x);`	valid	invalid
`inv x(x, , z);`	valid	valid
`inv x(x, , );`	valid	valid
`inv x(x, y, );`	valid	valid

The three invalid HAC examples are invalid because the definition of `inv` requires exactly three port arguments, where only fewer were given.

The above examples show a declaration with port connection in one statement. HAC allows declarations and port connections to be decoupled, so the following examples would be legal:

```
bool i, o;
inv x;
x(i, o);
inv y;
y(i, );
y(, o);
y( , );
```

6.2 Arrayed connections

HAC forbids the declaration of an array with port connections in the same statement[1]. For example, '`inv x[4](i, o);`' is invalid. Given an array, one should declare the array up front and then connect the ports in a loop.

```
bool i[4], o[4];
inv x[4];
(;j:4:
  x[j](i[j], o[j]);
)
```

[1] The rationale behind this decision is to disambiguate sparsely indexed declarations and arrayed connections. Most likely, such statements would form as the result of an error in translating from CAST.

7 Expressions

Expressions in HAC depart somewhat from CAST style and follow from ANSI C grammar more closely. Here we list a few differences.

The equality operator in CAST was `=`, but C and HAC use `==`. However, use of the deprecated `=` operator is allowed if the expression is wrapped in parentheses, e.g. `(x=y)`.

Less-than '`<`' and greater-than '`>`' operators now need to be parenthesized to dismbiguate them from template parameter delimiters, See Chapter 9 [Templates], page 17. For example, in CAST, `[ X<Y -> ... ]` would need to be re-written in HAC `[ (X<Y) -> ... ]`.

Logical operators in CAST were `&` and `|`, but in C and HAC, they are respectively `&&` and `||`. This makes way for bitwise operators (on integers) `&` and `|`.

XOR (bitwise integer) is `^` in HAC.

Logical XOR (for booleans) current uses `!=` instead of `^`, however, overloading `^` might be added back at some point.

The standard boolean and integer operations are strictly type-checked. At compile-time, there is no implicit conversion between `pbool` and `pint` values. For example, one cannot use an integer expression as a boolean, whereas in C, the integer is implicitly compared against 0.

Explicit casting operators may be added in the future.

Function expressions may be added in the future.

8 Loops

8.1 Loop instantiations and connections

In CAST, the an example of a loop statement in the meta-language might look like:

```
<i:N:
  inv z[i];
  x[i] = y[i];
>
```

The equivalent in HAC would look like:

```
(;i:N:
  inv z[i..i];
  x[i] = y[i];
)
```

The main difference is the use of parentheses instead of angle brackets and an extra semicolon operator. Notice that the declaration of `z` in the loop uses an explicit range to declare each sparse instance in the collection (Chapter 5 [Arrays], page 9). (It is highly recommended to keep declarations outside of loops where possible, leaving only connection statements inside loops.)

Like in CAST, loops may be nested arbitrarily deep and may be nested with conditional bodies.

8.2 PRS Loops

Do *NOT* write PRS bodies inside loops, the semantics are currently undefined. The PRS sub-language includes a similar loop syntax, described in Section 11.1 [PRS Loop Rules], page 21.

9 Templates

9.1 Template Basics

CAST supported parameterized definitions and types, also known as templates. HAC supports templates using C++-like template syntax. Templates are useful for defining a family types with highly regular differences. Such generalizations often occur in definitions whose port sizes or implementations vary enumerably or trivially. The most notable differences from CAST templates and HAC templates is the use of angle brackets around template arguments and parameter declarations, and the grammatic location of the template signature[1].

In CAST, a parameterized definition might look like this:

```
define foo(int N) (node[N] x) { }
```

The equivalent definition in HAC would be:

```
template <pint N>
defproc foo (bool x[N]) { }
```

CAST and HAC alike support parameter-dependent template parameters.

```
define array()(int N, D[N]) { }
```

would be written in HAC as:

```
template <pint N, D[N]>
defproc array() { }
```

Definitions are not the only templatable entities. Section 10.1 [Typedef Templates], page 19, discusses how typedefs can be templated in HAC.

9.2 Template Instantiations

In CAST, templates were instantiated by passing parameter values in parentheses, e.g. '`e1of(2)`'. In HAC, template parameters are passed in angle brackets, e.g. '`e1of<2>`'.

9.3 Default Parameters

See the HAC Language Specification.

9.4 Relaxed Templates

See the corresponding chapter of the HAC Language Specification document for more details. TODO: texinfo document link.

Relaxed templates allow certain parameters of a type template to *vary* within an aggregate collection (with the same base name).

Update: this has been implemented in the compiler.

[1] This originates from C++ template syntax.

10 Typedefs

Type aliases or *typedefs* were not supported in CAST, but are worth mentioning as a new feature of HAC. Like in C, typedefs are a mechanism for giving user-defined names to an existing type. (TODO: discuss the benefits of style.) If one really wanted to use `node` and `bool` as the same type, one could write: '`typedef bool node;`' and use `node` interchangeably with `bool`.

The real benefit is being able to bind definitions templates to new definitions that just forward template arguments to underlying types.

In the library '`channel.hac`', we see the following example:

```
template <pint N>
defproc e1of (bool d[N], e) { ... }

typedef e1of<2> e1of2;
```

This declarations defines type `e1of2` to be an alias to the complete type `e1of<2>`. In CAST, `e1of(2)` and `e1of2` are different definitions and hence, could not be equivalent types. Connecting them required connecting their public port members, which was an inconvenience when mixing template types with non-template types. More common examples can be found in the library '`env.hac`'.

10.1 Typedef Templates

Typedefs themselves may be templated, as best illustrated by the following example:

```
template <pint N, M>
defproc matrix(bool x[N][M]) { ... }

template <pint L>
typedef matrix<1, L> row;

row<3> a_row_of_length_3;

template <pint H>
typedef matrix<H, 1> col;

col<3> a_col_of_height_3;

template <pint N>
typedef matrix<N, N> square;

square<2> a_2x2_square_matrix;
```

The typedef template feature is useful for binding selected parameters of highly generalized template definitions to conveniently reduce the number of parameters.

Q: Can typedef templates be defined with default parameter values?

11 PRS

The syntax for normal production rules in the PRS language is the same as in CAST and HAC. See the PRS chapter of the Language Reference '`hac.pdf`'.

(Look at examples in the source '`test/parser/prs`'!)

11.1 Loop Rules

In CAST, loops of rules could be written like:

```
<i:N:
  ~x[i] | ~z -> y[i]+
>
```

The enclosing loop syntax is slightly different in HAC. The above example would be re-written as:

```
(:i:N:
  ~x[i] | ~z -> y[i]+
)
```

Again, rule loops may be nested arbitrarily deep. The notation `:N:` is also equivalent to `:0..N-1:`.

PRS-bodies may now appear inside instance-scope loops. The previous example could also be written as:

```
(;i:N:
prs {
  ~x[i] | ~z -> y[i]+
}
)
```

11.2 Conditional Rules

In CAST, conditional production rules could be written inside PRS bodies as follows:

```
prs {
[pred ->
  x -> y-
]
}
```

If the predicate *pred* evaluates true during the unroll compile phase, then the body is processed, otherwise it is skipped. In HAC, the syntax remains unchanged apart from the syntax of PRS expressions in the predicate. HAC also allows an optional else clause in the conditional body:

```
[pred ->
  x -> y-
[] else ->
  z -> y-
]
```

where in CAST, one had to explicitly write the predicates complement in a separate conditional to achieve else-semantics.

In CAST and HAC, conditional PRS bodies could appear inside instance-level conditionals and loops, such as:

```
[pred ->
  prs {
    ...
  }
]
```

Conditional bodies and loops in the PRS may be nested inside each other arbitrarily deep.

HAC conditionals also support sequential else-if constructs and else-clauses. For example,

```
[ expr1 ->
[] expr2 ->
[] expr3 ->
[] else ->
]
```

Only the first clause in the sequence with a true guard will be unrolled and expanded. If none are true, then all clauses are skipped. The `else` clause is optional.

11.3 Loop Expressions

In CAST, one could generalize an operator expression in the following manner:

```
<&i:N: x[i] >  -> y-
```

The LHS expression is the conjunction (AND) of nodes `x[0]` through `x[N-1]`. HAC also provides an equivalent construct:

```
(&:i:N: x[i] )  -> y-
```

The other operator which may be used in a loop expression is `|` (OR).

NOTE: expression loops whose range is empty (e.g. i:0:) are yet undefined, and the current compiler implementation rejects them at unroll time.

11.4 Rule Macros

Please read this section carefully.

Macros in PRS may be shorthand for other expanded rules or they may mean something different that isn't expressible in the PRS base language. CAST provided some built-in macros into its PRS language, which have been relocated into HAC's `spec` language (Chapter 13 [Spec Directives], page 27). For example, given '`exclhi(x,y)`' in CAST-PRS (not to be confused with CAST-spec's `exclhi`), the result of `cflat` would direct `prsim` to force nodes `{x,y}` to be exclusive high at all times. Namely the `exclhi`, `excllo` PRS macros have been renamed as `mk_exclhi` and `mk_excllo` *spec* directives. Q: were there any other PRS macros in CAST?

HAC has added support for emulating unidirectional pass-gates: `passn` and `passp`. The interpretation of these macros is tool-dependent.

Developers may define their own PRS macros by following the examples in the source code. Generalized macros also support a syntax for taking parameter values as arguments in addition to instance references. (We refrain from getting into that for this document.)

11.5 Rule Attributes

(I don't know the grammar for rule attributes in CAST.) In HAC-PRS, production rules may be tagged with an arbitrary number of attributes.

The most common example is the `after` delay attribute, which was written in CAST:

```
after 100  p -> q+
```

which would now be written in HAC:

```
[after=100]  p -> q+
```

Attributes appear as a semicolon-delimited, square-bracket-enclosed list of key-value pairs prefixing the rule. An example of multiple attributes:

```
[after=100;weak=1]  p -> q+
```

Attributes that appear before a loop-enclosed rule will apply to all iterations of the looped rule. The expressions in attributes may be parameter-dependent or induction-variable dependent, resulting in different values per iteration.

11.6 Sized Literals

Status: supported in syntax, but not used in any back-ends yet.

11.7 Miscellaneous

The HAC language no longer supports the `env` sub-language.

12 CHP

The CHP language, based on Hoare's CSP, is slightly different in HAC than in CAST.

12.1 Lexical Conventions

In HAC, `]%` is a token for "end probabalistic selection," thus, if the dividend of a modulus expression is indexed (ending with a `]`), then an extra space is required. For example, '`x[i]%2`' must be written as '`x[i] %2`'.

12.2 CHP Grammar

Nondeterministic selections are delimited by `:` in HAC, whereas they were delimited by `|` in CAST.

In CAST, send and receive actions were written as: '`X!x`' or '`Y!z`', but in HAC, send and receive arguments must be enclosed within parenthesis like function call arguments: '`X!(x)`' or '`Y!(z)`'. Rationale: syntactic consistency and disambiguation with the concurrency operator (comma).

HAC supports receive actions that do not write to any variables. '`Y?`' acknowledges channel `Y` without writing its values.

12.3 New Features

Compile-time meta-parameter repetitive expansions: see the HAC Language documentation's CHP chapter.

Includes loops for concurrency, sequence, and selection statements.

13 Spec Directives

The `spec` sub-language in HAC is a generalized version of the `spec` bodies in CAST. See the SPEC language chapter in the '`hac.pdf`' language reference.

Spec directives may be parameterized in HAC, and arguments may be passed in groups.

13.1 Porting from CAST

This section describes some common uses of spec-directives in migrating from CAST to HAC.

`CHECK_CHANNELS` production rules may be replaced with `exclhi` and `excllo` directives, which are used by `hacprsim` for built-in checking of mutual exclusion at run-time.

Forced exclusive high/low rings, which used to be *inside PRS* bodies, were declared as `exclhi` and `excllo`. In HAC, they are now the `mk_exclhi` and `mk_excllo` spec directives.

14 Legacy Tools

The old CAST tools all started with a source processor called `cflat` which translated top-level instances into a text stream of the flattened representation to be fed into subsequent tools in the toolchain. In this section, we describe how to migrate away from some of the tools or use the HAC front-end as a backward-compatible replacement for `cflat`.

The replacement, `hackt cflat` or `hflat`, features options similar to those of the original `cflat`. Instead of running '`cflat -<mode> <castfile>`' to produce flattened output, one now can now run '`hflat <mode> <objfile>`', where *objfile* is a compiled HAC object file. For example, '`cflat -prsim foo.cast`' would now be run as '`hflat prsim foo.haco`'.

Running `hflat` with no arguments will produce a list of the various present modes and format flags available as command-line options. All the formats used by the legacy tools come as named presets. Further fine-tuning of the output format can be controlled by individual '`-f <flag>`' command-line options.

14.1 PRS Simulation

To produce production rules suitable for simulation with the old `prsim` simulator, simply invoke: '`hflat prsim <objfile>`', where *objfile* contains compiled top-level instances and definitions. If the object file is not already created and allocated, the flattener will automatically do so (possibly catching and reporting errors from the later compile phases) before producing the flattened output. The flattener produces a list of all instantiated production rules and connections in human-readable text, which can be redirected to a file or piped straight into the old `prsim`.

There is also a completely rewritten production rule simulator named (you guessed it) `hackt prsim`, or just `hacprsim`. It emulates the behavior of the old `prsim` and provides new features. There (is, will be) a separate document for the new version, coming to a documentation directory near you. Very little documentation is required because there is a help system built into the program. A list of all commands with one-line descriptions can be browsed by running '`hacprsim -h`' with no other arguments. In the interpreter, help for any command or category can be viewed by typing '`help <command>`' at the prompt.

14.2 LVS

There was once a wrapper script named `cast2lvs` which flattened a single instance of a given type in the top-level, ignoring all other top-level instances. This is particularly useful for being able to LVS definitions and cells hierchically from the leaf-cells up, facilitating efficient layout verification. The new mechanism to emulate `cast2lvs` is to use: '`hflat lvs -t <type> <objfile>`', where *type* is the name of the complete type (with template arguments, if applicable), and *file* is the object file containing the compiled definition to unroll. The object file need not contain any top-level instances, which means it does not need to be unrolled. Top-level instances are simple ignored with the '`-t type`' option is used. Recommendation: redirect the resulting output to a '`.lvsprs`' file or pipe it straight into the old `lvs` program.

14.3 CHPSIM

For the CHP simulator, consult the '`hacchpsim.{info,pdf,ps}`' manuals.

Concept Index