The HACKT Object File Format

David Fang

September 27, 2009

Contents

Chapter 1

Introduction

The purpose of this document is to describe the intermediate file format used by the HACKT. DISCLAIMER: This is a *draft* of the proposed format. Before you panic, all the details for *how* objects are written-out and reconstructed are implemented through the HACKT's persistent object manager (POM) library. The POM provides a simple clean functional interface and masks all the details. Some of those details (for the maintainers) are described in the Persistence chapter of "Fang's C++ Utility Belt", a separate document.

HACKT is designed to unify the information associated with a design through a consistent interface. Every design flow has numerous tools to perform synthesis, analysis, and optimization. The information required and generated by each tool often overlaps with information handled by other tools. With HACKT, we provide a mechanism for passing and storing arbitrary data between tool invocations.

The first objective of establishing this format is to provide derivative tools access to all information in the full design hierarchy, enabling manipulation of information in definitions and individual instances. A lesser objective of the file format is to minimize the amount of stored data, i.e. eliminate redundantly redundant redundancy.

To resolve: For all structures described in this document, what debug information should be stored (source file, line, offset)? This could potentially encumber the object file a lot...

Chapter 2

Skeleton

The object file has a header and a body. The header summarizes the objects contained in the body. The header is managed by the POM.

2.1 Header

Proposal: version number.

The first item in the header is an integer (4B) that declares the number of 'objects' are contained in the file, which is the number of header entries that are listed in the header. (This is the number of objects that are dynamically allocated on the heap during the POM's reconstruction phase.)

Most of the remainder of the header contains a sequence of N header entries, one per object. Each header entry is composed of the following fields:

1. (8B) Object type code, currently an 8-byte code uniquely identifying the class or structure type.
2. (1B) Sub-type code, further refinement of the type.
3. (4B) Offset of the start this object (first byte).
4. (4B) Offset of the end this object (last byte).

(Yeah, I know, technically, the object's size is sufficient for reconstruction... detail of implementation.)

The first (0th) object entry corresponds to the null object, described below. The object code for the null-type is 8-bytes of 0s or null-characters, the sub-type is 0, and both offsets are also 0.

The remaining objects are normal. After the last object entry in the header, there's a hard-coded `0xFFFF` whose sole purpose is alignment checking. The offset after the alignment marker marks the end of the header and the start of the body. The offsets associated with each object entry correspond to the position of each objects binary data *relative to this position.*

2.2 Body

As mentioned before, the first and 0th object in the body is actually a null object. Since its start and end offsets are both 0, no space is actually used to store it — it doesn't exist.

The remainder of the body is just binary data segmented according to the offsets in the header. After initial object allocation, each object calls its own binary translator on its segment of data to reconstruct itself.

The only other special object is the body is the 1st (real) object. This is the *root object* for the entire file, returned by the POM upon completion of loading. The next chapter describes the format of the root object.

2.3 Footer

Optional, not yet implemented. May contain checksum, signature, or history of program invocations, etc...

Chapter 3

The Root Object

(The root object is of type `ART::entity::module`.) This chapter describes the format of the root object from which all other subordinate objects may be reached.

The module format is summarized as follows:

1. string: an optional module name
2. pointer: to the global (mother-of-all) namespace. This contains non-sequential data for the design. Namespaces are described in Chapter 4.
3. bool: whether or not the module is fully unrolled (Is this usable by tools past the front-end?)
4. sequence of pointers: sequential instance management data, used primarily by the front-end compile phases. This is described in Chapter 8.

And that's all there is to it at this level. (Why did I reserve an entire chapter for this?)

Chapter 4

Namespace Format

This chapter describes the contents of namespace objects. Namespace objects are recursive, they may contain other namespaces, arbitrarily deep.

Each namespace contains the following data:

1. string: name of the namespace
2. pointer: back-reference to parent namespace
3. local symbol table to objects belonging to this namespace, described in Section 4.1.

4.1 Symbol Table

The symbol table for each namespace is just a map of names to objects (pointers) that belong to the namespace. (The implementation is really irrelevant to the format.) Each entry of the symbol table is just a pointer to the subordinate object belonging to the namespace.

But wait, where are the keys used to map or sort each object? The key is stored in each object belonging to the namespace. Upon reconstruction, the deeper objects are reconstructed before their map entry in the parent symbol table is re-established. This eliminates the need to store the same strings twice.

4.2 Members

Aside from deeper namespaces, other namespace objects may include type definitions and physical instances. Type definitions are described in Chapter 5. Physical instances are described in Chapter 6.

Chapter 5

Definitions and Types

NOTE: This chapter is very incomplete.

All definitions types contain two (possibly empty) lists of template formals: a strict parameter list and a relaxed parameter list. The template parameter list format is summarized in Section 5.1.

All definitions start with the following data:

1. string: name
2. pointer: back-reference to parent namespace
3. Template parameters (Section 5.1)

The sub-types of definitions extend the fields:

Data type definitions

1. ...

Channel type definitions

1. ...

Process type definitions

1. Sequence of public ports
2. Symbol table: same as described in Section 4.1
3. Sequential instance management for unrolling (internal connections), as described in Chapter 8

Definitions contain member layout maps, one *per unique tuple* of template arguments. Before type-checking is complete, the list of layout maps will be empty. After unrolling (before expansion), each definition will be populated with layout maps for each unique tuple of parameters that it is invoked with. (Does NOT exhaustively check all possible values — impossible.)

5.1 Template Parameters

Template parameters contain two sequences (lists) of pointers of parameters, that may include formal values, or formal type parameters (not yet implemented). (All formal parameters in these lists are also registered with the local symbol table.)

5.2 Process Ports

Ports are implemented as a sequence (list) of pointers to physical instances (collections) that are publicly accessible to the outside. (All referenced instances here are also registered with the symbol table.)

5.3 Typedefs

Typedefs are aliases to defined types. Typedefs may also contain template parameters. Typedefs' symbol tables should contain only template formal parameters and public port parameters, i.e. no additional local symbols.

1. string: name
2. pointer: to parent namespace
3. Template parameters
4. pointer: to base type reference (Section 5.4)

Incidentally, typedefs are sub-typed for processes, channels, data-types.

5.4 Type References

Type references contain:

1. pointer: to a base definition
2. pair sequence of template arguments that complete the type (first pair of strict arguments, second pair for relaxed arguments)

Incidentally, type references are sub-typed for processes, channels, data-types. There is no different in content except that the base definition's type is restricted to the respective subtype.

5.5 Member Layout Maps

NOTE: Not yet implemented, in progress.

Each time a template definition is 'used' with fully specified parameters (a complete type), it is fully type-checked with those parameter values. If the type-check passes, it creates a layout map, whose purpose is to translate member references into data offsets. (Those offsets are used to *address* the appropriate (physical) subinstance.

A member layout map entry consists of a string (name of member) and an offset (integer).

(Subtype layout maps by process, channel, data? Depends on how physical instances are structured, see Chapter 6.)

Chapter 6

Instances

Virtual vs. physical instances

Hierarchical names handled by the front-end correspond to virtual instances. Virtual instances are an indirection to physical instances. Multiple virtual instances may alias to the same physical instance. (Nomenclature: virtual instances are currently called instance aliases in the compiler implementation — get the names straight!)

Only after all connections have been unrolled and established are the unique physical instances created. This step of unique instance creation is the final step of the front-end compiler.

Implementation: Concern: this expanded form is only useful to a subset of the tools, such as simulators and analyzers. The expanded form is going to be large, which is undesirable for the tools that don't need it. Consider: The final unique creation phase should be tool-specific, but provide a library function for it.

6.1 Instance Collections

Every instance name may refer to either a single (scalar, 0-dimensional) virtual instance, or a higher-dimension array (up to 4) of virtual instances. When we refer to instance collections, we also include the case for 0-dimensional instances.

The contents of an instance collection is summarized as follows:

1. pointer: back-reference to parent namespace
2. string: name of this instance collection
3. map of virtual instances that are *actually unrolled* during the unique creation phase. Each individual virtual instance contains the index with which it is associated. The virtual instances sequence is stored as a list of pointers.

Prior to the creation phase, the map of index-instance pairs may be empty.

In addition, the following subtypes have additional fields stored with the instance collections:

- Process: contains an additional process type reference.
- Channel: contains an additional channel type reference.
- Data (excluding int and bool): contains an additional data type reference.
- int: contains a integer (4B) width parameter
- bool: nothing else

For instance collections that reside outside of definitions, the types referenced by the instance collections must be resolved to canonical definitions i.e, they may not reference typedefs. Instance collections residing inside definitions may reference typedefs.

NOTE: the type reference contained by the collection must have strict template arguments specified, but need not have relaxed template arguments specified. If the relaxed arguments are specified, then they apply to all instance members of the collection, and the members are forbidden from supplying relaxed template arguments. If the relaxed arguments are not specified, then the instance members (see Section 6.2) *must* supply relaxed template arguments to complete their types.

Scalars instance collections simply replace the sequence of virtual instance with a single virtual instance. The scalar collection's type reference must specify relaxed template arguments, if applicable.

6.2 Virtual Instances

All virtual instances have the following data in common:

1. multidimensional index used to associate with the collection of which it is a member (only if non-scalar)
2. pointer: alias connection to the next instance of its kind[1].
3. pointer: to the unique physical instance (spawned only after the unique creation phase)
4. pointer: back-reference to the collection of which this is a member

---

[1] Connections are maintained as circular linked lists of virtual instances of the same type. This way, all aliases are reachable from each other. If it is not aliased to another instance yet, then the pointer refers to this object, a self-reference.

5. (proposed): relaxed template parameters, which may be different for each member of the collection (with the same strict parameters, of course)

The only distinction for the various subtypes of virtual instances is that the pointers are statically typed to their own respective type. (Processes point to process collections, physical processes, etc.)

6.3 Physical Instances

These are only created by the expansion phase of the compiler, which follows the final type-check phase. (There is currently a debate whether the expansion phase should be done by default.)

6.4 Instance References

Instance references contain the following data:

1. integer (4B): an index into the instantiation statement list indicating which statements were in effect (visible) at the point of reference. (This is for catching errors that cannot be statically resolved.)
2. pointer to the (subtype-specific) referenced (virtual) instance collection
3. pointer to list of index pointers, where the individual index pointers may be pointers to single integers or integer ranges.

6.4.1 Member Instance References

Member instance references contain the following data:

1. pointer to the enclosing instance reference (generic)
2. pointer to the (subtype-specific) instance collection that is a member of the enclosing instance reference. (Membership must be name-checked.)

Chapter 7

Parameter Values

The chapter describes the format of values and value collections. `pints` and `pbools` have value semantics, and thus, are treated differently than physical instances, which have alias semantics.

7.1 Value Collections

Value collections are very similar in structure to virtual instance collections, described in Section 6.1.

1. pointer: back-reference to parent namespace or definition
2. string: name of this value collection
3. map of value entries that are created and evaluated during the unroll phase. Each map entry contains:
   (a) the index (key) with which it is associated
   (b) value stored
   (c) instantation flag (whether or not the entry has truly been created[1])
   (d) valid flag (whether or not value has actually been set)

For `pints`, the stored value is an integer (4B), and for `pbools`, the boolean value and two boolean flag are packed together into one byte.

7.2 Value References

Value references are classified as expressions. The content of value references is structured like virtual instance references.

---

[1] This is an artifact of HACKT's current implementation, and may be removed in the future.

1. integer (4B): an index into the instantiation statement list indicating which statements were in effect (visible) at the point of reference. (This is for catching errors that cannot be statically resolved, and can only be resolved at unroll time.)
2. pointer to the (type-specific) referenced value collection
3. pointer to list of index pointers, where the individual index pointers may be pointers to single integers or integer ranges.

Chapter 8

Sequential Instance Management

Appendix A

Common Primitives

This appendix describes the formats of common sub-structures of the HACKT Object File Format.

A.1 Strings

The binary format for strings is simply the length of the string (4B), followed by the contents of the string. Null-termination on write is not necessary because the length is known. Upon reconstruction however, the reader will automatically add null-termination.

A.2 Pointers

All pointers (references to other persistent objects) are translated by the POM into an integer (4B), the index of the object in the body section of the object file. The obvious limitation of this is that there's a hard limit of 2^{32} manageable objects. Talk to me when you exceed this limit. For the gory details on this subject, please refer to the entire "Fang's C++ Utility Belt" document.

A.3 Sequences

The standard format of any sequence (list, array, etc.) is the number of elements (4B), followed by the data for each element in sequence.

A.4 Expressions

The storage for all forms of expressions follow straight from the language itself.

A.4.1 Booleans

Single boolean values are stored as one byte each. Groups of bools, however may be compacted into the same byte (bit-fields), as long as the reader and writer for a particular set of bools is consistent.

A.4.2 Integers

Integers are stored as 4 bytes each, unless noted otherwise.

A.4.3 Floating-point

Haven't added support for floating-point values yet.

A.4.4 Indices

An index is just a sequence of pointers to integer-expressions. Currently, the size limit is 4, the limit of number of dimensions supported by the compiler.

A.4.5 Ranges

A range is a pair of pointers to integer-expressions.

A.4.6 Binary expressions

All binary expressions are stored as:

1. char: representing the operation
2. pointer to left operand expression
3. pointer to right operand expression

A.4.7 Unary expressions

All unary expressions are stored as:

1. char: representing the operation
2. pointer to operand expression

Index