

Verrazano Design Specification

Rayiner Hashem

June 6, 2007

1 Introduction

1.1 Intention

The purpose of this document is to specify, in detail, the design of the “verrazano” C++ binding generator. It attempts to explain the syntax and semantics of the input file, the structure of the program, the intermediate representation used by the frontend and backend, the design of the utility library and the overall semantics of bindings generated by the program. It is intended that this document be updated continuously as the program evolves, and serve as a reference guide for developers interested in writing language-specific backends or modifying the program itself.

1.2 Notation

In situations where it is necessary to represent objects or algorithms in the abstract, a truncated form of Common Lisp is used. For example, the *defclass* syntax is used to represent class types, with the initial ‘defclass’ element omitted for brevity. It is expected that despite the truncation, the semantics of the pseudo-code will be clear to Common Lisp programmers.

1.3 Theory of Operation

Verrazano is an additional tool invoked during the build process for a program, analogous to an interface definition language (IDL) compiler or pre-processor. Given an input file specifying one or more C++ header files, it

generates a module that allows code in the target language to access the functions, constants, and classes declared in those headers. For Lisp, for example, given an input file, the program generates a package containing foreign function interface (FFI) declarations. Conceptually, the input file is analogous to a package definition in which C++ code and objects are exported instead of Lisp code and objects.

It is important to note that some C libraries are using preprocessor macros (`#define`'s) even for their API functions (e.g Posix). These macros can potentially expand to different C functions on each compilation of the C file(s). To handle such situation in a platform independent way, a C wrapper library must be generated. This C wrapper code must be then compiled and loaded on each installation. Although Verrazano can be used to generate FFI definitions for the generated C wrapper lib, it can not help much in creating the wrapper C code itself.

1.4 System Structure

Verrazano is composed of several components: a frontend, one or more backends, and an utility library. The frontend is responsible for processing input files and generating an intermediate representation (IR) of the C++ headers referenced in those files. It is also responsible for simplifying the IR to make it more easily digestible by the backends. The backend for a given target language is responsible for taking the simplified IR and generating from it appropriate FFI declarations. The utility library exists to provide stub-code generated by backends with common functionality, in particular routines for constructing virtual function tables (v-tables) for specified classes.

2 Frontend

2.1 Input File

Conceptually, input files are analogous to a *defpackage* declaration in Lisp. The input file specifies the headers to parse as well as options for controlling the form of the generated bindings. It is illustrative to demonstrate a sample file. This file shows how to export a few functions from the SDL library. One of the functions is annotated to signal to the backend that the return parameter should be converted into a Lisp string.

```
(defbinding "sdl-library"
  (nicknames "sdl")
  (include "SDL.h")
  (export "SDL_Init" "SDL_Quit" "SDL_GetError")
  (override "SDL_GetError" ("string" ("void"))))
```

As the listing shows, the input file is in prefix form, and besides the initial *defbinding* element, can contain four constructs. The semantics of these constructs are listed below.

- **nicknames:** One or more strings signifying the nicknames for the generated module. For the Lisp backend, these nicknames are passed unchanged to the *defpackage* declaration.
- **include:** A list of header files that should be imported into the binding. This list can be constructed assuming the default C lookup rules.
- **export:** A list of symbols that should be exported by the binding.
- **override:** A declaration annotating functions or methods with additional information about their parameters. A declaration takes the form of a symbol string and a list of the form *(return-value (arg1 arg2... argn))*. In the declaration, the real types of the function can be replaced with symbols that present additional information about the parameter.

It should be noted that symbol names in the input file should be fully-qualified in C++ syntax. Thus, a class *foo* within the namespace *bar* should be referred to as *foo::bar*. In order to improve the ease-of-use of the program, it would be highly desirable to allow regular expressions in the symbol strings, to allow specification of multiple overrides with a single statement.

2.2 C++ Header Parsing

Verrazano uses GCC-XML to parse C++ header files. GCC-XML uses a slightly modified version of the GCC C++ parser to generate an XML file describing the C++ constructs defined in a header file. The program parses this XML file to generate its IR of the header file. The GCC-XML file format itself is not described here, as it is specified in [1]. A few notes about the format are in order, however. The XML file format is slightly unusual in

that it is not explicitly hierarchical. Constructs that are explicitly nested in C++ header files, such as member fields, appear in a flat node-space in the XML format. Hierarchical information inherent in the C++ header is specified not by nesting of XML nodes, but rather by attributes in certain nodes named “context” or “member”. It is important to keep this feature of the file format in mind when attempting to understand how the program handles GCC-XML’s output files.

2.3 Intermediate Representation

In the frontend, GCC-XML files are parsed into an intermediate representation (IR) to make the information easier to manipulate in later stages of the program. The IR is a directed graph, with types represented as nodes and interactions between them represented as edges. The IR exists in the frontend as a set of objects. The class hierarchy for these objects is described below.

- (annotable ()(annotations)): The base class, allows arbitrary annotations to be added to IR elements.
- (node (annotable)(name)): Represents nodes in the IR.
- (type (node)()): Represents C++ types.
- (namespace-type (type)()): Represents the *type* of a namespace, which is defined as the set of definitions contained within the namespace. There is no particularly good reason to define namespaces this way instead of having a dedicated *namespace* node, but having a *namespace-type* nicely parallels how functions are handled in the IR.
- (fundamental-type (type)()): Represents fundamental C++ types like *int* or *float*.
- (pointer-type (type)()): Represents pointer and reference types.
- (array-type (type)(size)): Represents arrays of a given length.
- (enum-type (type)()): Represents a C++ enumeration.
- (struct-type (type)()): Represents simple structures and classes with no member functions.

- (union-type (type)()): Represents simple unions.
- (class-type (type)()): Represents structures and classes with member functions or base classes.
- (function-type (type)()): Represents prototypes of functions.
- (alias-type (type)()): Represents typedefs.
- (edge (annotable)()): Represents edges in the IR.
- (returns (edge)()): A *returns* edge from node *A* to node *B* signifies that objects of type *A* return objects of type *B*.
- (receives (edge)(name)): A *receives* edge from node *A* to node *B* signifies that objects of type *A* receive objects of type *B* as parameters.
- (defines (edge)(name)): A *defines* edge from node *A* to node *B* signifies that the definition of *A* contains the definition of *B*.
- (allocates (edge)(name value)): An *allocates* edge from node *A* to node *B* signifies that objects of type *A* allocate objects of type *B* as part of their in-memory representation.
- (extends (edge)(name value)): An *extends* edge from *A* to *B* signifies that the definition of *A* is an extension of the definition of *B*.

This class hierarchy is designed with an eye towards making it easy to manipulate the IR and to generate declarations using graph traversals. In particular, it is designed so that the declaration corresponding to any node can be emitted in response to a multi-method dispatch on the type of the node, the type of the parent node, and the type of the edge between them. One of the implications of this design is that there are no explicit nodes corresponding to function, namespace, or variable declarations. Instead, these constructs are represented using an edge between two type nodes. For example, function declarations are specified by an *allocates* edge from a *namespace-type* node to a *function-type* node. Declarations of namespaces, global variables, and structure member fields are represented in a similar manner.

2.4 Simplification Passes

Before being sent to the backend, the IR is subject to a series of simplification passes in order to reduce the number of special cases that must be considered in the backend. The specific simplification passes applied to the IR is dependent on the backend. It is conceivable, for example, to have a source-to-source backend that required no simplification passes. In real backends, it is expected that one or more simplifications will be necessary. A list of possible such simplification passes is presented below.

- **Collapse Namespaces:** Most languages that will potentially have backends have no feature analogous to C++'s arbitrarily nested namespaces. While Lisp packages could be used to emulate some uses of nested namespaces, a design goal of verrazano is to keep a 1:1 mapping between an input file and an output Lisp package. This simplification pass collapses namespaces and changes the names of members to eliminate conflicts. It is expected that the precise naming behavior will be specifiable as a user-provided policy.
- **Collapse Nested Definitions:** Most FFIs do not allow nesting of structure and type declarations as C and C++ do. For example, whereas a C++ structure can define substructures, no FFIs allow this sort of nested definition. This pass generates fully equivalent definitions for such cases by pulling child definitions out of their parents and renaming them to eliminate conflicts. As with the previous simplification pass, it is expected that the naming policy will be a user-specifiable parameter.
- **Name Anonymous Nodes:** Many nodes in the IR may not have names, as a result of the underlying C++ constructs. For example, nested structures used only to define a single field will generally be anonymous. In many cases, to preserve the proper semantics, the backend will have to generate declarations for these anonymous nodes. This pass makes proper generation of these declarations possible by assigning names to all anonymous nodes in the IR. These names need not be human-readable, since they will never be referenced directly by the programmer, so any algorithm that assigns unique names will suffice.
- **Convert Enumerations:** Not all the FFIs verrazano could potentially target have a way to define enumerations. This pass generates fully

equivalent definitions for enumerations by replacing them with appropriately-named sets of constant variables.

- Convert Arrays: Not all FFIs have a way to specify that a foreign type is an array. This pass generates equivalent definitions for arrays by converting them to bare pointers.
- Promote Structs: The C++ conception of a structure differs from verrazano's conception of a structure. In C++, *struct* and *class* are equivalent except for the default visibility of members. In verrazano's IR, structures are plain records, with no member functions, no members that contain member functions, and no base classes. Basically, they are the same as a *struct* in C. This pass looks for *struct-type* nodes in the IR that violate these restrictions and converts them to *class-type* nodes.
- Demote Classes: This pass is similar to the one above, except that it looks for *class-type* nodes that meet the aforementioned restrictions and demotes them to *struct-type* nodes.
- Mark Simple Classes: There is a subset of classes that can be handled with a much less code than is required to implement full class semantics. Specifically, classes that define no virtual member functions and have no bases that define any virtual member functions can be treated as simple structures and their member functions can be treated as regular functions that accept a pointer to the object as their IRst parameter. This pass looks for such classes and adds an annotation to their *class-type* node signaling to the backend to perform this optimization
- Mark Final Classes: Classes that will not be subclassed from Lisp code also require fewer declarations than is required for classes that exhibit full class semantics. This pass identifies such classes, using information provided by the user in the input file, and marks the appropriate *class-type* nodes in the IR.
- Annotate Memory Layout: While FFI's generally have some ability to discover the memory layout of C code, for example, being able to find the offsets of member fields in structures, none have the full abilities

required for C++ bindings. This pass annotates the IR with information about the in-memory layout of objects, specifying, for example, the v-table offsets of member functions.

- **Sort Definitions:** Not all FFIs are insensitive to the order of declarations in the source file. Since GCC-XML does not define any particular ordering of definitions within its XML files, a traversal of the IR will not necessarily create FFI declarations in dependency order. For FFIs that cannot tolerate this, a topological sort of the IR could be used to arrange nodes in their proper order. However, the IR is not necessarily acyclic, because the underlying C++ definitions may be cyclic. Problematically, GCC-XML does not emit nodes for forward declarations, so there is no way to eliminate these cycles before constructing the IR. This pass will use the FFI's mechanism for forward declarations in conjunction with heuristic approaches for roughly topologically-sorting graphs to order the IR before generating FFI declarations.

2.5 GCC-XML to IR Mapping

The mapping from the GCC-XML input file to the frontend's IR is for the most-part straightforward. This section specifies how each construct in the XML input file maps to nodes and edges in the IR. To make the mapping easier to follow, the list below is in the order node types are declared in the GCC-XML DTD. Note that since the IR is a directed graph, it is important to distinguish between an edges extending to a node and and edges extending from a node.

- **Argument:** Represented as a *receives* edge from a *function-type* node to a *type* node.
- **ArrayType:** Represented as an *array-type* node with an *extends* edge to a type node.
- **Base:** Represented as an *extends* edge to a *class-type* node.
- **Class:** Represented as a *class-type* node.
- **Constructor:** Represented like a function with an annotation marking it as a constructor.

- Converter: Represented like a function with an annotation marking it as a converter.
- CvQualifiedType: Represented as annotations in the base *type* nodes.
- EnumValue: Represented as a *allocates* edge from an *enum-type* node to a *fundamental-type* node of type integer with an annotation marking it constant.
- Enumeration: Represented as a *defines* edge from a *namespace-type* node to an *enum-type* node.
- Field: Represented as an *allocates* edge from a *class-type* node to a *type* node.
- Function: Represented as an *allocates* edge from a *namespace-type* node to a *function-type* node.
- FunctionType: Represented as a *function-type* node.
- FundamentalType: Represented as a *fundamental-type* node.
- Method: Represented as an *allocates* edge from a *class-type* node to a *function-type* node.
- MethodType: Represented as a *function-type* node.
- Namespace: Represented as an *allocates* edge to a *namespace-type* node.
- OperatorFunction: Represented like a function with an annotation marking it as an operator.
- OperatorMethod: Represented like a method with an annotation marking it as an operator.
- PointerType: Represented as a *pointer-type* node.
- ReferenceType: Represented as a *pointer-type* node.
- Struct: Represented as a *struct-type* node.
- Typedef: Represented as an *alias-type* node.

- Union: Represented as an *allocates* edge from a *namespace* node to an *union-type* node.
- Variable: Represented as a *allocates* edge from a *namespace* node to a *type* node.

3 Common Lisp backend

3.1 Universal Foreign Function Interface (UFFI)

During the initial development of *verrazano*, UFFI [2] will be the target backend for Common Lisp. UFFI is an FFI abstraction designed for high performance and portability to multiple Lisp implementations. As it is currently specified, it contains enough functionality to enable significant access to C libraries. However, by itself it is not powerful enough to allow full access to C++ libraries. In particular, it lacks callbacks and the ability to call through function pointers. These features are all necessary to enable a high-level of integration between C++ libraries and Common Lisp programs. Since *verrazano* will be developed in conjunction with the Hello-C project, it is expected that as Hello-C stabilizes it will become the main backend for *verrazano*.

3.2 Intermediate Representation

The backend does not define its own intermediate representation, since it does not do any manipulation of UFFI constructs once they are generated. Instead, it does a straightforward traversal of the IR, directly generating UFFI declarations in the process.

3.3 IR to UFFI Mapping

The generation of UFFI declarations from the IR is done using a depth-first traversal of the IR graph. Each stage of the traversal considers a pair of nodes and the edge between them, and generates declarations based on that consideration. The mapping between (node-type, edge-type, node-type) combinations and UFFI constructs is detailed below. It should be noted that many such combinations have no semantics, either because they cannot occur,

are removed by simplifications prior to reaching the backend, or because they require no response. These combinations are omitted from the list below.

- (*namespace defines (pointer-type or array-type)*): *def-foreign-type*.
- (*namespace allocates fundamental-type:constant*): *def-constant*.
- (*namespace defines enum-type:constant*): *def-enum*.
- (*enum-type allocates fundamental-type:constant*): Fields in the *def-enum* declaration.
- (*namespace defines struct-type*): *def-struct*.
- (*struct-type allocates type*): Fields in the *def-struct* declaration.
- (*namespace defines union-type*): *def-union*.
- (*union-type defines type*): Fields in the *def-union* declaration.
- (*namespace allocates type*): *def-foreign-var*.
- (*namespace defines class-type*): This case requires more code generation than the other cases and is described in the next section.

3.4 Binding to C++

Binding to C++ code is often identical to binding to C code. A significant amount of functionality in C++ libraries can be accessed using regular UFFI *def-function* declarations. There are a number of cases, however, in which C++ code must be called differently from C++ code, and these are listed below. Note that this list is based on the GCC C++ ABI [3].

- Non-static Non-virtual Methods: The calling convention for these functions is identical to the regular C calling convention, except they take a hidden first parameter — a pointer to the C++ object the method is called on. This type of function can be handled simply by making this parameter explicit then generating a *def-function* for the method.

- Virtual Methods: The calling convention for these functions is identical to that of a non-static, non-virtual method called through a pointer. The target of the call can be determined by indexing into a v-table accessible from the object pointer. This type of function can be handled by creating a stub function that performs the v-table lookup and calls through the resultant pointer.

3.5 Integration with the Common Lisp Object System (CLOS)

Integrating the C++ type hierarchy into the CLOS class hierarchy is a desirable and sometimes necessary feature. Many C++ libraries, for example Qt, depend on client programs subclassing library types. Integration of C++ and CLOS types can be achieved using a combination of Lisp CLOS proxy objects, generic methods, stub functions, and a utility library capable of generating v-tables for CLOS subclasses derived from C++ classes.

For each non-final C++ class, the UFFI backend generates a CLOS proxy type. Objects of this type have no members other than a pointer to a native C++ object. For each method in the class, the backend generates a CLOS generic method that uses one of the techniques described in the previous section to access its underlying C++ method. Additionally, it uses the utility library to create a v-table for the class. It initializes this table to be a copy of the C++ class's v-table, then for each method overridden in Lisp code, it initializes the corresponding table element to point to a stub method that transfers control to the generic method dispatcher. This scheme allows calls in both directions, from Lisp to C++ and from C++ to Lisp.

Having given a rough overview of how C++ and CLOS integration could be achieved, it is interesting to touch upon what extensions to UFFI would be necessary to allow this integration. The following list summarizes the features necessary in UFFI for the aforementioned design to work properly.

- Calling functions through a pointer: This feature is necessary to access C++ virtual functions, as well as to take advantage of some of the features of regular C Libraries.
- Exporting Lisp functions to C code: This feature is necessary not only for allowing the subclassing of C++ classes from Lisp code, but to take advantage of any C library that requires client code to supply callbacks.

To maximize the ease-of-use of verrazano-generated C++ bindings, the following features would be desirable in the target FFI.

- Subclassing of foreign types: It significantly eases the work of the backend, when handling C++ class declarations, if it is able to create native types that are subclasses of foreign types.
- Control over type conversions during foreign calls: Integration of C++ libraries into Lisp code is significantly increased if the Lisp code can work with native types rather than foreign types. Allowing the backend to specify to the FFI that C++ strings should be converted to Lisp strings, or that C++ vectors should be converted to Lisp vectors would eliminate the need for the backend to generate stub functions to implement these conversions.

4 C++ Utility Library

Manipulating of v-tables is an ABI-specific, rather than backend-specific task. Thus, it is desirable to define a utility library for the manipulation of v-tables that can be shared by all verrazano backends. The utility library must be able to read a serialized version of the frontend IR and from it generate virtual function tables for a given class. It must also be able to take the virtual function table for an existing class and modify certain table entries to point to backend-specified stub functions. These tasks can be accomplished in a fairly straightforward manner using algorithms specified in the ABI for the target compiler.

References

- [1] <http://www.gccxml.org/files/v0.6/gccxml-2004-11-19.dtd.txt>: Interim file format specification for GCC-XML 0.7, CVS version.
- [2] <http://uffi.b9.com/manual/>: Universal Foreign Function Interface Reference Manual.
- [3] <http://www.codesourcery.com/cxx-abi/abi.html>: IA64 C++ ABI, used as the cross-platform GNU C++ binary interface.