Web技術文書からのFFI自動生成に関する実践（2012年8月6日版）

小野田 武朗[1]　菅谷 みどり[1,2]　倉光 君郎[1,2]

概要：近年，多くのスクリプト言語にて，Foreign Function Interface (FFI) が利用されている [7][5]．FFI とは，あるプログラミング言語から，多言語で実装されたコンポーネントを使用するためのインターフェースである．FFI を利用するには，言語間での型の違いや関数呼び出し規約の違いを吸収するグルーコードの作成が必要となる．この際，グルーコードの開発・保守作業を手作業で行うと，FFI の対象とするコンポーネントの API が大量にある場合，膨大な開発コストがかかるという問題がある．また，それらのインターフェイスが変更になった場合，それに対応するための保守作業にもコストも無視できない．一方近年では，Web 上に開発者向けの情報を提供することが一般的となっている．Web の情報は更新や，修正が容易であることから，最新の開発者向け情報（技術情報）が提供されていることが多い．さらに，多くの関数ライブラリやクラスライブラリなどの FFI の対象となる文書は HTML 文書として提供されている．れらの技術文章は，規則性をもった記述がなされていることが多いことから，我々は，この点に着目し，これらの HTML の技術情報から情報抽出を行い，グルーコードを自動生成する手法を提案するした．また，グルーコードの生成に正確な型情報が必要となることから，本研究では，静的型付けスクリプト言語 Konoha に対する FFI 生成の実験を行い，その評価をおこなった．

A Practice for Automatic Generation of FFI from Web Technical Document (version 2012/8/6)

Takeo Onoda[1]　Midori Sugaya[1,2]　Kimio Kuramitsu[1,2]

Abstract: Recently, scripting language provide rich functions via their foreign function interface. Manualy developping or maintaining FFI glue code is a huge cost. On the other hand, most of big libraries has a official API document in format of HTML. These HTML documents describes API interface in details, and tagged with type information. In this paper, we present our proposing method of automatic generation of glue code from these HTHML documents. Additionaly, we evaluated our method in its correctness, cover-rate with OpenGL library.

1. はじめに

近年，多くのプログラミング言語にて，Foreign Function Interface(FFI) が利用されている．FFI とは，あるプログラミング言語から，他言語で実装されたコンポーネントを利用するためのインターフェースである．開発者は，FFI を用いることにより，様々な言語で実装された既存のライブラリを使用し，その機能・性能を享受することができる．特に，スクリプト言語から，C 言語などのコンパイル型言語のライブラリを利用できるメリットは大きく，多くの試みが行われている．

通常，FFI を用いるためには，言語間での異なる型定義や，関数の呼び出し規約の違いを吸収するグルーコードの作成が必要となる．この際，グルーコードの開発・保守作業を手作業で行うと，FFI の対象とするコンポーネントの API が大量にある場合，膨大な開発コストがかかるという問題がある．この問題に対して，SWIG（Simplified Wrapper and Interface Generator）などのグルーコードの

[1] 横浜国立大学大学院工学府
Graduate School of Engineering, Yokohama National University
[2] 日本科学技術振興機構/CREST
Japan Science and Technology Agency/CREST

© 2012 Information Processing Society of Japan

自動生成を行うツールが提案されている．SWIG は，ヘッダファイルからグルーコードを自動生成するツールであり，既に様々な言語で利用されている．しかしながら，自動生成の際に，対象となるライブラリ側のマクロやインクルードヘッダがあるケースなどでは，生成されるグルーコードの品質が悪く，結局手作業で追修正が必要となる．

一方，近年では，Web 上で開発者向けの情報が提供されることが一般的となっている．Web の情報は，更新や修正が容易であることから，最新の開発者向け情報（技術情報）が提供されていることが多い．さらに，多くの関数ライブラリやクラスライブラリなどの FFI の対象となる文書は HTML 文書として提供されている．これらの技術文章は，規則性をもった記述がなされていることが多い．このことから，我々は，この点に着目し，これらの HTML の技術情報から，情報抽出を行い，グルーコードを自動生成する手法を提案した．この際に，グルーコードの生成に正確な型情報が必要となることから，本研究では，静的型付けスクリプト言語 Konoha に対する FFI 生成の実験を行い，その評価を行った．

本論文の貢献は以下の通りである．

(1) Web 技術文書から情報抽出を行い，グルーコードを生成するための FFI コードジェネレータを Konoha スクリプト上に設計，実装した．
(2) 実装したコードジェネレータを用いて，Qt ライブラリ [1] の FFI を自動生成し，評価を行った．
(3) SWIG と Konoha のコードジェネレータの比較を行い，提案の有効性を示した．

本論文の構成は以下の通りである．第 2 節では既存のグルーコード自動生成手法による問題点を述べる．第 3 節では，提案するグルーコード自動生成手法について説明し，その手法にしたがったグルーコードジェネレータの設計を述べる．第 4 節では，提案するグルーコードジェネレータの実装，第 5 節では，提案手法に対する実験・評価を行い，第 6 節で，関連研究について述べる．最後に，第 7 節にて，まとめと今後の課題について述べる．

2. 問題定義

本節では，いくつかの代表的なプログラミング言語を例として，FFI 開発の重要性，現状，問題点を述べる．ここでは，C 言語の 3 次元レンダリングライブラリである OpenGL[2][9] を例にあげて説明を行う．

2.1 OpenGL

OpenGL（Open Graphics Library）は，グラフィックスハードウェアのアプリケーションプログラミングインターフェース（API）である．OpenGL では，API が C 言語の関数やマクロとして提供されている．OpenGL の機能を利用しようとする場合，一般的には API を直接使用できる C 言語から利用することが多い．

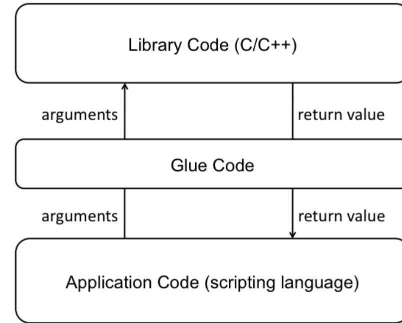

図 1　FFI アーキテクチャ
Fig. 1　Architecture of FFI

2.2 FFI の重要性

アプリケーション開発の実装効率を向上させるために，C 言語ではなく，スクリプト言語から OpenGL の API を利用する場合について考える．

スクリプト言語上で OpenGL のライブラリ API を再現する方法の一つとして，ライブラリをスクリプト言語により再実装することがあげられる．しかし，この方法では C 言語によるライブラリの実装内容を全てスクリプト言語に移植する必要があり，開発コストが非常に高い．

スクリプト言語上でライブラリ API を再現するもう一つの方法として，スクリプト言語から OpenGL が提供している C 言語の API を直接利用する方法があげられる．この方法では，各 API ごとに言語間の違いを吸収するコード（グルーコード）を作成することにより，既存のライブラリを使用することができる．グルーコードを用いた FFI の構造を，図 1 に示す．これは，ライブラリの再実装コストの低減と，実装言語の性能を生かすことができることという２つの利点を含んでいる．このように，FFI を利用して他言語のライブラリ API を呼び出すことにより，効率的なアプリケーション開発が行うことができるという点で，FFI は大きなメリットがある．

2.3 グルーコード開発の現状

次に，グルーコード開発の現状について述べる．FFI の実現のためには，言語間での違いを吸収するためのグルーコード作成が必要となる．グルーコードの具体的な役割として，言語間で異なる型定義や，関数の呼び出し規約の違いの変換などがある．従来，グルーコードの開発は手作業で行われることが多かった．しかし，手作業での開発には，作成や保守作業にかかるコストが大きい．これに対して，グルーコード開発コストを低減するための，グルーコード自動生成ツールが提案されている．SWIG（Simplified Wrapper and Interface Generator）[4] などの，C/C++言

© 2012 Information Processing Society of Japan

語のヘッダファイルからグルーコードの自動生成を行うグルーコード自動生成ツールが近年では広まっている．

2.4 グルーコード開発の問題点

2.4.1 コストの問題

グルーコード開発を手作業で行う場合，その開発コストが課題となる．例として，OpenGL を C 言語以外のプログラミング言語から FFI により用いる場合について考える．

OpenGL ライブラリに含まれる API の数は ver2.1 時点で約 460 個である．ライブラリサポートの観点から，すべてのライブラリ API に対するグルーコードを作成したい．ひとつひとつの API では，引数の型変換，関数呼び出し，エラーハンドル，返り値の型変換を含むグルーコードが２０行程度で収まるとしても，すべての API に対してのグルーコードは１万行を越えるコード量となる．このように大規模な，ライブラリの API すべてをサポートするためには，膨大な量のグルーコードの作成が必要になる．また，OpenGL はバージョン数も多く，グルーコードの追加，修正のコストを避け，再利用性を高めるためにバージョン毎にコードを書くと考えれば，さらに作業量は増えるため，手作業でこれらすべての作業を行うのは，コストの点から大きな問題となる．

2.4.2 品質の問題

上記した開発コストの問題を解決するために，OpenGL のようなライブラリについては，グルーコード自動生成ツール（グルーコードジェネレータ）を用いてグルーコード開発を行うことがある．ジェネレータを用いて作成したグルーコードにおいて，もととなるライブラリの提供 API 数に対するカバー率を，自動生成されたグルーコードの品質と呼ぶことにする．一般的なグルーコードジェネレータの構成として，C/C++言語のヘッダファイルから API を抽出し，抽出された API に従ったグルーコードの生成を行う．このような構成をとる既存のツールとして，SWIG があげられる．しかし，ライブラリ API の抽出を C/C++言語のヘッダファイルから行うには，以下のように単純な抽出が困難な箇所がある．

関数マクロ 関数がマクロとして定義されている場合，ソースコード上からは，ライブラリの開発者がどのような引数・返り値の型を想定しているのか読み取ることが難しい．

include 文の処理 ライブラリのヘッダファイル内で include 文が取り扱われている場合，それを展開するかどうかが問題となる．展開を行わなければ全てのライブラリ API を網羅できないことがあるが，その場合，標準 C ライブラリのヘッダファイルなど，ライブラリに無関係なヘッダファイルまで展開されてしまう．

ライブラリが提供しない関数，クラスの識別 ライブラリ内部処理のみで使用され，外部への提供を意図していない関数，クラスにおいては，グルーコードを作成する必要がない．ヘッダファイルにより，外部公開するものと，外部公開しないものの区別がはっきりとされていればよいが，そうでない場合，ソースコード中からグルーコードの作成をするかどうかの判断をすることが難しい．

C++パーサ 名前空間の解決や，クラス階層の解析など，C++のヘッダファイルの解析を完全に行うパーサの作成が困難である．

3. 提案

本節では，我々の提案するグルーコードジェネレータの設計，および，FFI ライブラリ作成の対象とした，静的スクリプト言語 Konoha[8][3] について説明をする．

3.1 提案の概要

我々は，グルーコードを作成するために，既存のツール等で提案されているヘッダを利用した方法では十分に品質をあげたグルーコード生成の自動化が難しいと考えた．そこで，近年，ライブラリの技術文章が HTML 文書として公開されていることに着目し，これら HTML 化された技術文章（Web 技術文章）からの情報抽出をもとにグルーコード生成を試みることとした．対象とする Web 技術文書は，ライブラリの作成者が提供する，API の記述，説明がなされているものである．

Web 技術文書を利用する利点は２つある．１つ目は，Web 技術文書ではライブラリの提供する API の情報が構造化されて記述されているため，機械的な情報抽出が行いやすいことである．HTML 文書は，すべての情報にタグ付けがされており，これはより情報抽出を簡便にする．構造化された Web 技術文書の例として，OpenGL では funcdef というクラス属性を持つタグのあとに，関数のインターフェースが記述されている．２つ目の利点は，Web 技術文書は，開発元が提供していることからソースコードと同等の信頼性があると考えられることである．グルーコードの品質向上をめざすには，情報抽出をする情報源の正確さについての信頼性が必要となる．ヘッダファイルを用いる手法では，ヘッダファイル自身がライブラリのソースコードの一部であり，情報源としての信頼性は高いといえる．これに対し Web 技術文書では，それ自身はライブラリ本体と別の存在であるが，開発元が提供しているものに限り，ソースコードと同等の信頼性を持つと考えて，以下で説明する提案手法に用いた．

3.2 グルーコードジェネレータの設計

我々は，第２節で述べた問題を解決するために，Web 技術文書からの情報抽出によるグルーコードジェネレータの設計をした．本ジェネレータでは，C/C++で実装された

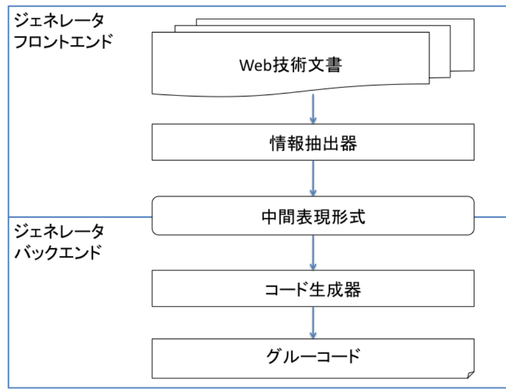

図 2 ジェネレータアーキテクチャ
Fig. 2 Architecture of a generator

ライブラリを，静的型付けスクリプト言語である Konoha から使用するためのグルーコードを生成する．我々の設計したグルーコードジェネレータは，情報抽出部とコード生成部の 2 つのコンポーネント，およびその間でのデータをやりとりするための中間表現から構成される．本節では，Konoha の FFI 機構の説明，およびジェネレータ設計の詳細について述べる．図 2 に，設計をしたジェネレータのアーキテクチャを示す．

3.2.1 Konoha FFI 機構

Konoha は，静的な型付けされたスクリプト言語であり，C/C++で実装された関数・クラスを用いるための FFI 機構が用意されている．Konoha では，静的な型付けを行うため，プログラムの実行時には適切な型付けが行われていることが保証されている．そのため，FFI を用いる際に，グルーコード内で型チェックをする必要はない．本研究では，FFI における型安全性についての議論を省くため，言語的に FFI の型安全性が保証されている Konoha を用いてジェネレータの作成を行った．

3.2.2 Web 技術文書からの情報抽出器

Web 技術文書は，HTML 文書により構成される．そのため，グルーコードの生成に必要な情報の抽出は，HTML 文書のパースをして行う．我々はこれを情報抽出器と呼ぶ．情報抽出器は，HTML 文書中の特定のタグに囲まれた情報を抽出・解析をするというごくシンプルな設計にした．本コンポーネントは，Web 技術文書から抽出した情報を，3.2.3 項で述べる中間表現として出力をする．

3.2.3 中間表現形式

ジェネレータ内部で情報抽出部とコード生成部を分けることにより，情報抽出部の切り替えによる情報源の変化への柔軟な対応や，コード生成部の切り替えによる多言語のグルーコード生成への対応が可能であるため，我々は抽出した情報を，一度，中間表現に落とした．ジェネレータ内部での機能のモジュール化をするにあたり，情報抽出部とコード生成部とのデータのやり取りをする形式を定める必要がある．クラスがメソッドを持ち，メソッドが引数

型	Konoha	C/C++
整数	Int	int, long, short
少数	Float	float, double
文字列	String	char *
配列	可変長	固定長
オブジェクト	クラス	構造体，クラス

表 1 C/C++と Konoha 間での型変換規約
Table 1 principle of type conversion between C/C++ and Konoha

などの情報を持つといった，入れ子構造を容易に表現でき，またデータへのアクセスのしやすいデータ構造として，XML や JSON などのデータ構造があげられる．本研究では，JSON を採用し，抽出された情報を JSON 形式で保持することとした．

3.2.4 コード生成器

コード生成器は，中間表現を入力として，グルーコードを生成・出力する．プログラミング言語ごとに，言語として扱うことのできる型が定められているため，FFI を用いる際には，言語間での適切な型の変換を行う必要がある．本ジェネレータでは，型変換を行うためのタイプマップをコード生成器内に保持しており，表 1 にその対応表を示す．

C++言語はクラスベースのオブジェクト指向プログラミング言語であり，クラスごとの継承関係が存在する．そのため，C++言語で実装されたクラスを Konoha から違和感なく用いるためには，Konoha 上でクラスの継承関係を再現する必要がある．ただし，C++言語では多重継承をサポートしているが，Konoha では単一継承のみをサポートしている．このため，C++言語で多重継承が行われているクラスは，Konoha ではそのうちの一つのクラスのみを継承するようクラスの継承関係を変更して再現するよう設計した．

3.2.5 Web 技術文書を利用するリスク

Web 技術文書から情報抽出を行うことによるリスクもいくつか考えられる．以下にその例をあげる．

バージョンの不一致 情報抽出を行う Web 技術文書での記述対象となるライブラリと，インストールされているライブラリのバージョンの不一致が起こりえる．自分の環境に合わせた，適切な Web 技術文書を使用することが必要となる．

記述ミス Web 技術文書内に記述されているライブラリ API において，クラス名や関数名などに記述ミスがある場合，Web 技術文書から抽出された情報に誤った情報が混じってしまう．

4. 実装

本節では，設計したグルーコードジェネレータの実装について説明する．グルーコードジェネレータの実装には，Konoha と Python を用いた．

© 2012 Information Processing Society of Japan

4.1 情報抽出器の実装

Web 技術文書では，HTML の構成が一般化されておらず，ライブラリ単位での情報抽出器の作成が必要となってくる．情報抽出器の処理の流れは，以下のようになる．

(1) HTML 文書内でのライブラリ API の記述されかたに規則性がないかをチェックする
(2) 規則性を発見できたら，それに従ってタグの検索，情報の抽出を行う
(3) 情報された抽出をパースして中間表現を生成する

OpenGL における抽出の例をあげる．OpenGL では，すべての関数のインターフェースは funcdecl というクラス属性をもつタグのあとに記述されている．ひとつの関数に対してひとつの JSON のオブジェクトを作成し，関数名，引数の型，引数の名前，返り値の型を要素に持たせた．

また，Qt における抽出の例をあげる．Qt は C++のライブラリである．ひとつのクラスに対してひとつの JSON のオブジェクトを作成した．さらに，ソースコードからは抽出が困難なシグナル・スロットについての情報抽出を行い，クラスに対応する JSON オブジェクトに情報を持たせた．

4.2 コード生成器の実装

Konoha では，グルー関数のインターフェースとして以下の knh_Fmethod 型にあわせた形で定義される．

```
void (*knh_Fmethod) \
    (CTX ctx, ksfp_t *sfp, long _rix);
```

ここで，ctx は言語ランタイムの管理情報（Context）であり，sfp は Konoha の言語ランタイムが持つスタックの構造（Konoha スタック）の先頭を示すポインタ，_rix は返り値が置かれる Konoha スタック上のインデックスを示している．

4.2.1 関数呼び出し規約の変換

グルーコードの動作の流れは，以下のようになる．

(1) Konoha からの引数の取得
(2) C/C++ライブラリ関数のコール
(3) Konoha への返り値の受け渡し

この流れにしたがって，グルーコードの生成を行っていった．Konoha から引数を取得するためには，Konoha スタック（sfp）にアクセスする必要がある．グルーコードが呼びだされたとき，sfp には，引数が適切な形でのせられている．よって，グルーコードの生成を行いたい C 言語の関数の引数の型と個数がわかっていれば，グルー関数内で型チェックを行う必要がない．よって，グルー関数内での引数の取得は，C 言語の関数の引数の数と型により，一意に定まり，それに応じた形でコードの生成を行う．本ジェネレータでは，関数・メソッドを，グローバル関数，クラスメソッド，static クラスメソッド，コンストラクタ，に分類し，グルーコードの生成を行った．この分類による違いは，グルーコード内部での関数・メソッドの呼び出し方法である．グローバル関数では関数を直接呼び出すが．クラスメソッドではクラスインスタンスを取得してから関数の呼び出しを行う．また，static クラスメソッドでは，クラスインスタンスの取得が不要で，スコープ解決演算子をつけてメソッドの呼び出しを行う．コンストラクタは，new 演算子をつけて関数の呼び出しを行う．Web 技術文書からの情報抽出を行う時点で，関数の宣言より，関数の分類分けによるフラグを立て，グルーコードを自動生成する際に，フラグに応じた呼び出し方で，コードの生成を行う．

Konoha への返り値の受け渡しは，引数の取得と同様に，C 言語での型から Konoha の型へ変換し，sfp を用いて受け渡す．

グルー関数の動作の流れは，以下のようになる．

(1) Konoha からの引数の取得
(2) C/C++ライブラリ関数のコール
(3) Konoha への返り値の受け渡し

この流れにしたがって，グルー関数の生成を行っていった．

4.2.2 クラス構造の再現

C++言語はクラスベースのオブジェクト指向プログラミング言語であり，クラスごとの継承関係が存在する．そのため，C++言語で実装されたクラスを Konoha から違和感なく用いるためには，Konoha 上でクラスの継承関係を再現する必要がある．ただし，C++言語では多重継承をサポートしているが，Konoha では単一継承のみをサポートしている．このため，C++言語で多重継承が行われているクラスは，Konoha ではそのうちの一つのクラスのみを継承するようクラスの継承関係を変更して再現するよう設計した．

4.2.3 オブジェクトのラッピングと GC

C/C++言語での構造体・クラスは，Konoha では kRawPtr 型として扱われる．Konoha の kRawPtr 型は C 言語の構造体で，メンバとして，オブジェクトの情報を保持するヘッダ（kObjectHeader）と，C/C++言語の構造体・クラスを保持するためのポインタ（void*）を持つ．以下に，kRawPtr 定義の疑似コードを示す．

```
struct kRawPtr {
  kObjectHeader h;
  void *rawptr;
}
```

malloc や new などで，メモリのヒープ領域に構造体・クラスが確保されたとき，kRawPtr の rawptr にその先頭アドレスを格納する．Konoha では Bitmap を利用した MSGC が実装されている．Konoha の GC で kRawPtr 型オブジェクトが破棄される際に，rawptr の先に格納された構造体・クラスを free する必要がある．また，GC 時に free をしたくないものに関して，MSGC におけるマークを恣意的に行う処理を記述する．例として，コールバック関数用の情報などがあげられる．以上の GC 処理を，グルーコードとし

© 2012 Information Processing Society of Japan

て自動生成する．

5. 評価

提案するグルーコードジェネレータの実験・評価を行う．

5.1 コストの評価

提案するグルーコードジェネレータを用いて，OpenGL の Konoha バインド作成を行った．結果を表 2 に示す．また，比較として，SWIG を用いて OpenGL の Python バインド作成を行った結果を，表 3 に示す．提案手法と既存手法（SWIG）による差は大きくでなかったが，提案手法により既存手法と同等以上のグルーコード品質を得ることができた．ただし，SWIG のインターフェースファイル５行でよいのに対し，提案手法での情報抽出器の作成には１７５行のコストがかかった．また，生成できたグルーコード行数を比較すると，SWIG により生成されたグルーコードの方が，はるかに行数が多い．これは，Python のグルーコード内では引数の型チェックを行なっているためだと考えられる．

5.2 品質の評価

提案するジェネレータを用いて，C++ライブラリである Qt の Konoha バインド作成を行った．Qt では，第 2.4 節でとりあげた，グルーコードの品質向上における問題点を全て含んでいる．表 4 に結果のまとめを示す．

第 2.4 節であげたヘッダファイル解析の問題点をもつライブラリにおいて，Web 技術文書の解析を行うことにより，グルーコード生成に一定の成果が見られた．抽出できた FFI の API 数と，生成した FFI の API 数が異なるのは，Konoha でのグルーコードを生成における，言語上の制限があったためである．C++でのテンプレートを用いたクラスや，オーバーロードをしている関数など，グルーコード作成が困難な API についてはグルーコード作成を行わなかった．

5.3 考察

既存手法であるソースコードからのグルーコード生成と比べて，提案手法である Web 技術文書からのグルーコード生成では，コストの面では少し劣るが，品質の面では，同等以上の結果を得ることができ，情報抽出の精度の高さが示せた．

対象とした OpenGL のバージョン	2.1
OpenGL の提供 API 数 （ドキュメント内に記述されている API 数）	370
抽出できた API 数	
生成した FFI の API 数	583
カバー率	
グルーコード行数	1851
FFI 生成を実行するまでのコスト (情報抽出器のコード行数)	146

表 2 提案するジェネレータによる OpenGL バインド
Table 2 OpenGL binding by proposed generator

対象とした OpenGL のバージョン	2.1
OpenGL の提供 API 数 (gl.h 内に記述されている API 数)	461
抽出できた API 数	457
生成した FFI の API 数	457
カバー率	99 %
グルーコード行数	20907
コード生成対象としたヘッダファイル	gl.h
FFI 生成を実行するまでのコスト （インターフェースファイルの行数）	5

表 3 SWIG による OpenGL バインド
Table 3 OpenGL binding by SWIG

対象とした Qt のバージョン	4.7
Qt の提供 API 数	17620
抽出できた Qt の API	17620
生成した FFI の API 数	7486
カバー率	100 %
グルーコード行数	262233

表 4 提案するジェネレータによる Qt バインド
Table 4 Qt binding by proposal generator

特に，Qt においては，Qt の独自機構である，シグナルとスロットについても正確な情報抽出が行えた．シグナルやスロットは，C++言語の組み込みの言語機構ではないため，単純な C++パーサでは情報の抽出が難しい．しかし，Web 技術文書を用いることにより，ライブラリ特有の機構についても，容易に情報抽出が可能であることが示された．また，情報抽出器の実装を少し変更すれば，C/C++言語以外の言語においても情報抽出の対象とすることができると予想され，Web 技術文書からの情報抽出は，非常に自由度の高いものであると言える．その反面，高い自由度を得るために，FFI 自動生成をするまでのコスト（Web 技術文書からの情報抽出のコスト）が，既存手法よりも高くなってしまいがちである．世の中にある Web 技術文書が，すべからく同じ形式で記述されていれば問題はないが，そのような統一された規格は現在のところ存在しないので，情報抽出器を，ライブラリごとにつくる必要があるからである．

容易にソースコードから API の抽出できない，Qt のような複雑な構造をもったライブラリに関しては，Web 技術文書からの情報抽出は，効果的な手法だといえるだろう．

6. 関連研究

既存の FFI に関する研究の潮流は２つある．１つ目は，我々のようにグルーコード自動生成に関するもので，上

© 2012 Information Processing Society of Japan

節であげたとおり既存ツールとしては SWIG や SIP が使われている．もう一つの潮流は，異なる言語間での安全な関数呼び出しを行う事に着目した研究である．Harren と Necula らは FFI を通じて呼び出される外部コンポーネントのソースコードを静的解析手法を用いて検査し，安全な外部関数呼び出しを実現するフレームワークである CCured[6] を提案した．また，Hirzel と Grimm らは C 言語と Java 言語の接続の際の安全性を高めるため，２つ言語を統合してプログラムを記述する Jeannie[7] を提案している．本研究では安全性については議論していないが，生成先の言語に複数対応するため，解析器と生成器を異なるコンポーネントとしてある．そのため，単純に２つの言語間での FFI よりも柔軟性があるといえる．

7. おわりに

本論文では，Web 技術文書から情報抽出を行うことによるグルーコードの自動生成手法を提案した．また，提案手法を用いた，グルーコードジェネレータを実装し，C のライブラリである OpenGL の Konoha バインディングの自動生成を行った．生成されたグルーコードを，既存のグルーコードジェネレータである SWIG と比較し，Web 技術文書からの情報抽出による情報精度の高さを示した．本稿執筆時点では，FFI 自動生成の対象言語が Konoha にしぼられている点や，グルーコードジェネレータの抽出機構の汎用性が低い点が問題点としてあげられる．今後の取り組みとして，FFI 自動生成の対象言語を Python や Ruby といったスクリプト言語へ拡張し，より実用的なツールの開発に取り組んでいきたい．また，Web 技術文書へ汎用的に用いることのできる，ライブラリ API 情報の抽出をより容易にする API の開発を進めていきたい．

参考文献

[1] : Qt - Cross-platform application and UI framework, http://qt.nokia.com/.

[2] : OpenGL - The Industry Standard for High Performance Graphics, http://www.opengl.org/.

[3] : KonohaScript, http://konohascript.org.

[4] Beazley, D. M.: SWIG: an easy to use tool for integrating scripting languages with C and C++, *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, Berkeley, CA, USA, USENIX Association, pp. 15–15 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1267498.1267513⟩ (1996).

[5] Furr, M. and Foster, J. S.: Checking type safety of foreign function calls, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pp. 62–72 (2005).

[6] Harren, M. and Necula, G. C.: Lightweight Wrappers for Interfacing with Binary Code in CCured, *ISSS*, pp. 209–225 (2003).

[7] Hirzel, M. and Grimm, R.: Jeannie: granting java native interface developers their wishes, *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pp. 19–38 (2007).

[8] Kuramitsu, K.: KonohaScript: static scripting for practical use, *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, New York, NY, USA, ACM, pp. 27–28 (online), DOI: http://doi.acm.org/10.1145/2048147.2048161 (2011).

[9] Woo, M., Neider, J., Davis, T. and Shreiner, D.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition (1999).

© 2012 Information Processing Society of Japan