kVerifier と SystemC

kVerifier and SystemC

小林憲次
Kenji Kobayashi

栃木県黒磯市阿波町 117-835　E-mail: kenji@nasuinfo.or.jp

ABSTRACT.　kVerifier is a C++/STL library that verifies C/C++ program codes generally. I apply kVerifier to Reed Solomon ECC simulation and STL/valarray modeling and SystemC circuit discription. I exemplify that kVerifier and STL/valarray modeling is effective in system design and the examination, and that the test vector at system design is available for the RTL design.
.

１．はじめに

kVerifier は C/C++ プログラムのテスト・検証・シミュレーション汎用的に行う C++ ライブラリです。テストに機能に特化したライブラリともみなせます。

テスト・データをテスト・ベクタ・ファイルとしてソースから独立させます。テストされるプログラムの関数や変数の型とアドレスを kVerifier に登録します。

kVerifier は時間進行を管理・制御します。時間に依存するプログラム動作のテスト・検証・シミュレーションも可能です。時間依存するテストを可能にしたことで、組み込みプログラムん回路記述プログラムのテストも kVerifier の対象とできるようにできました。

組み込みプログラムや回路記述などのテスト対象プラットフォームごとに kVerifier とのインターフェースを設けることで様々のプラットフォームでのテストを可能にします。回路記述のテストでは SystemC を対象としています。そのほかに汎用のライブラリを対象とするインターフェース、仕様記述スレッド・インターフェース、ワンチップ・マイコンなどの小さな RTOS k-uOS を用意しています。

この論文では最初に kVerifier の概要を説明します。次に STL/valarray による Galois 有限体の行列を述べ Rees Solomon コードについて説明します。そして valarray による Reed Solomon コードの記述を行い、それを SystemC の回路記述に変換します。仕様記述と回路記述でテスト・ベクタを共用できることを示します。

2. kVerifier が働く仕組み

kVerifier は登録された変数／関数の名前を使ってテスト・ベクタとアプリケーション・プログラムを協調動作させます。テスト・ベクタより入力変数のシーケンス・データを読んでプログラムを動作させます。動作結果をテスト・ベクタの出力変数の予定値と比較して確認します。

- 入力変数値を、その変数アドレスに設定して、プログラム・コードを動作させます
- アドレスを登録してある関数をテスト・ベクタに指定した引数で呼び出します
- プログラムが入力に応答した出力変数値の結果を予定値と比較・確認させます

この動作原理を表すブロック図を図１に示します。

プログラムの開始時に、変数のアドレスと名前と型、関数のアドレスと名前と関数型を kVerifier に登録しておきます。一方で、テスト・ベクタ・ファイルには、入力変数のシーケンス・データを変数の名前とデータ値を使って記述しておきます。関数の呼び出しシーケンスを関数名と引数値を使って記述しておきます。それらを受けてプログラムが変化させる出力変数のシーケンス・データを記述しておきます。kVerifier はテスト・ベクタの名前とデータ値を、登録してあった変数／関数と対応させます。入力変数を設定します。関数を呼び出します。出力を確認します

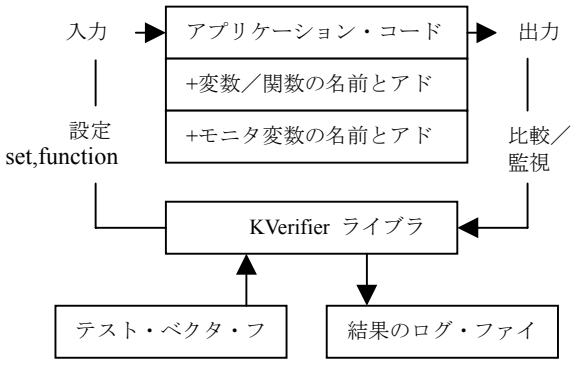

図１　kVerifier の動作原理

「結果のログ・ファイル」にはテスト・ベクタとアプリケーション・プログラムが動作した結果の全てが記録されます。テストでエラーが検出されたとき、ログ・ファイルをみれば、エラーが発生するまでのシーケンスが記録されています。プログラム開発途中のバグの多くは単純なものです。多くのバグはログファイルを見るだけで原因を推定できます。対策できます。ログファイルはデバッグ時に有効です。特定のデータのみを抜き出してタイム・チャート表示をするときにも使います。

モニタ変数として変数を登録しておくと kVerifier はその変数の変化を監視しつづけます。変化が発生すると、そのタイミングとデータをログ・ファイルに記録します。記録されたモニタ変数のデータはデバッグするときに有効な情報となります。

３．変数／関数の登録

図２ にベリファイ変数／関数インスタンス、モニタ変数インスタンスの生成登録テンペレート関数を示します。

```
 1 TcVerified<T>* tfNewVerified(T& rTAg
 2    , const string& crStrAg, long lgFlagAg = ios::hex)
 3
 4 TcVrfyMkNm<T>* tfNewVerified(T& rTAg, int inBtSlctAg
 5              , const string& crStrAg, long lgFlagAg = ios::hex)
 6
 7 TcVrfyFp<T>* tfNewVerified(T& rTAg, const string& crStrAg
 8              , double dbPrecisionRateAg)
 9
10 TcVrfyCntnr<T>* tfNewVfCntnr(T& rTAg, const string& crStrAg)
11
12 TcMonitored<T>*
13 tfNewMonitored(const T& crTAg, const string& crStrAg)
14
15 template<class R, class T> TcRgstFnctn<R,T>*
16 tfNewVfFnctn(R(*pfAg)(T), const string& crStrAg)
17
19 template<class Ty, class R, class T>
20 TcClRgstFnctn<Ty, R,T>* tfNewVfClFnctn
21          (Ty* pTy, R(Ty::*pfAg)(T), const string& crStrAg)
```

図３ ベリファイ／モニタ変数関数インスタンスの生成登録

テンペレート関数を使うことで、様々の変数型を少ない関数で登録できるようにしています。また変数型、関数型自体のの登録を可能にしています。

テンペレート関数の特別バージョンを活用して、複数のベリファイ／モニタ変数の登録関数を一つに見せるようにしています。現状のコンパイラで判別できない変数・関数について、テンペレート関数の名前を少しづつ変更しています。

19 -- 21 行目の は tfNewVfClFnctn(.) 、クラス・メンバー関数をベリファイ変数として登録するテンペレート関数です。クラス･メンバー関数を呼び出すときには this 引数が必要となるので、専用のインターフェースが必要になります。クラスのデータ・メンバーについては this 引数が必要ないので、クラスに属さないデータと登録関数を共用できます。

4 テスト・ベクタ

図４にテスト・ベクタの例を示します。テスト・ベクタには、プログラム動作の具体例を記述します。アルゴリズムは記述しません。

動作具体例の羅列としてのテスト・ベクタでは、変数を set すること、またはインターフェース関数を呼び出すこと function、または動作結果を確認すること compare の三つが基本要素です。set / function を使って入力条件を設定します。compare を使って、動作結果を確認します。set / function / compare が行われるタイミングをテスト・ベクタから指定できるようにすれば、大部分の動作具体例は記述できます。

```
1 ##     test vector example code
2 +0c    vlrGfStt          __set 3 6 10
3 +3uS port                __set 0x02
4 +1c    tesGfOperation __function mul 0x36 0x0f
5 +1c                       __wait
6 +0c    byResStt          __compare 0x2f
```

図４ テスト・ベクタ

図４で ‘#’ **で始まる** 1 行目はコメント行です。2 行目の最初の +0c はリセット解除の後 0 clock のタイミングで実行されることを意味します。リセット解除直後に vlrGfStt 配列変数に 3 6 10 の三つの数値を設定しています。

３行目では 3uSec 後に port 変数を 0x12 に設定しています。4 行目では 1 clock 後に tesGfOperation(.) 関数を “mul 0x36 0x0f” 引数で呼び出しています。 6 行目では動作の結果 byResStt 変数が 0x2f になっていることを確認しています。

5 行目の __wait は同期を取っています。tesGfOperation (.) 処理が終了したら __wait の次の 6 行目のテストを実行することを意味します。

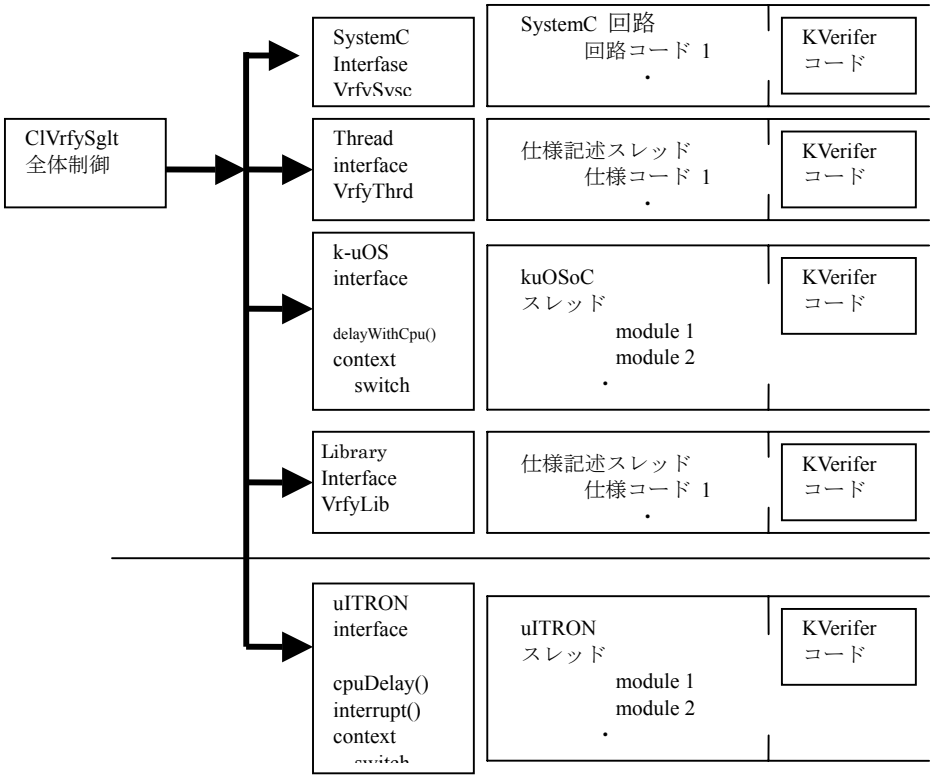

図５　回路、仕様、タスク全体の連携

5　回路、仕様、タスク記述の協調動作

回路も含めて、組み込みの全体を C/C++ 言語で記述できるようになってきました。kVerifier は回路、仕様、タスクの全体をテスト・ベクタを介在させて、シミュレーション実行・検証します

ワンチップ・マイコンの開発言語は C 言語が普通になりました。RTOS も C 言語で記述してあれば、ワンチップ･マイコンの組み込みプログラムを MS Windows などのクロス環境で動作させられます。現在では回路記述も SystemC などを使って C 言語記述ができます。kVerifier は、これらの C 言語で記述してあるプログラム・コードを連携させてシミュレーション実行・検証を行います。

現在の段階では kVerifier は以下のアプリケーションを連携動作させることを想定しています。

- 回路記述を行う SystemC
- RAM の少ないマイコンでのタスク記述を行う k-uOS
- C++/STL を使った並列記述を可能にする仕様記述スレッド ClVrfyThrd
- 代表的な RTOS -- uITRON

SystemC, RTOS タスクなどのアプリケーション・プログラムの動き方はベース・ソフトにより異なります。ベース・ソフトに合わせた kVerifier interface を用意し、アクティブにする kVerifier interface をダイナミックに切り替えることで、全体を協調動作させます。

k-uOS, uITRON interface に設けられている cpuDelay(delayAg) は CPU 消費時間をシミュレートするためのものです。この関数が呼び出されると、それを呼び出したスレッドの実行は中断されて他の kVerifier interface に CPU の使用を許します。シミュレーション時間が delayAg 時間だけ経過したあとに、delayCpu() の次のアドレスの制御が戻ってきます。

６．STL valarray と行列処理

STL は数値計算のための配列処理 valarray を用意しています。valarray は部分配列を操作する slice 機構を備えています。これを使えば行列演算も記述できます。valarray を使った行列演算ライブラリを Galois 有限体に適用して Reed Solomon codec を記述します。

valarray<T> は数学で扱う配列をプログラム記述するときに有効です。従来のプログラムでは for ループ記述が必要だったところを、 valarray<T> 演算で記述可能にします。valarray ヘッダ・ファイルに用意されている slice(.) が部分配列に限定した操作を可能にします。slice(.) は行列処理記述プログラム記述を簡単にします。

下のように記述することで valarray インスタンスを宣言・定義できます。

```
1 valarray(const T& valAg, size_t sztAg));
2 valarray(const T *pTAg, size_t sztAg);
3 T& operator[](size_t n);
```

１行目は長さが sztAg の valAg で初期化された配列をしています。２行目は配列 *pTAg で初期化された、sztAg 長さ配列宣言で。３行目の vlrAt[size_t n] により n 位置の配列要素の参照を返します。[.] operaotor によって配列要素を操作できます。

valarray では行列も配列として扱います。そして n 行、m 列など配列の一部を扱う下の slice を用意しています。

```
slice(size_t sztStartAg, size_t sztSizeAg, size_t sztStrideAg);
```

valarray[slize(.)] により部分配列を表します。sztStartAg は、元配列に対する開始位置です。sztSizeAg は部分配列の要素数です。sztStrideAg は部分配列を選択するときの開始位置からの跨ぎ幅です。

残念ながら STL には valarray があっても行列どうしの掛け算や逆魚列処理のライブラリまでは用意されていません。でも valarray<T> と slize(.) を活用すれば掛け算や逆行列の template ライブラリを記述できます。別のフィルタ処理向けのプログラムのを作りながら開発した汎用の template 記述した行列ライブラリがあります。このライブラリは下のような行列の積演算と Crout 方による逆行列演算処理を備えています。

```
template<class T> valarray<T>
mulMatrix(const valarray<T>& crVlrMatrixAgL
        , const valarray<T>& crVlrMatrixAgR, size_t sztAg=

template<class T>
class TcCroutGf{
    TcCroutGf(const valarray<T> crVlrAg)
    {
        SetMatrixLU(crVlrAg);
    }
    void SetMatrixLU(const valarray<T> crVlrAg)
    valarray<T> TransfVctByInv(const valarray<T>& crVlrAg)
}
```

７． Reed Solomon コード

n 次の Reed Solomon ECC(Error Collection Code) では下の g(X) 多項式を使って ECC を生成します。

$$g(x) \equiv (X+1)(X+\alpha).....(X+\alpha^n)$$

オーディオ CD では n=4 の下の Reed Solomon コードが使われています。

$$g(x) \equiv (X+1)(X+\alpha)(X+\alpha^2)(X+\alpha^3)$$
$$= X^4 + 0x0f * X^3 + 0x33 * X^2 + 0x78 + 0x40$$

$$SendData(X) = Data(X)X^n + ECC(X)$$
$$SendData(X) \bmod g(x) = 0$$
$$\Leftrightarrow SendData(\alpha^k) = 0 \;\; k = 0 \ldots n$$

送信データは GF(2^8)[X] 上の多項式 SendData(X) で表現できます。ECC を追加された送信データは

SendData(X) は g(X) で割り切れます。すなわち α^0,... α^n を代入すると 0 になります

また受信したデータ ReceivedData(X) に誤りがなければ ReceivedData(α^k) == SendData(α^k)== 0 となります。逆に Ｂ_j の位置に E_j のエラーが発生すると

$$S_k = \mathrm{Re}\,ceivedData(\alpha^k) \;\; k = 0 \ldots n$$

ReceivedData(α^k) = E_j* Ｂ_j* α^k != 0 となります。下の受信データから得られるデータ Sk をシンドロームとよんでいます。

受信データには複数のエラーが存在します。エラー位置 Bj エラー・データは Ej は複数存在します。このときのシンドロームは以下のようになります。

$$S_0 = \mathrm{Re}\,ceivedData(\alpha^0) = E_k\beta_k{}^0 + .. + E_0\beta_0{}^0$$
$$S_1 = \mathrm{Re}\,ceivedData(\alpha^1) = E_k\beta_k{}^1 + .. + E_0\beta_0{}^1$$

$$S_n = \mathrm{Re}\,ceivedData(\alpha^n) = E_k\beta_k{}^n + .. + E_0\beta_0{}^n$$

上の式はエラー位置については線形ではありません。上の式からでは行列の逆演算により求解ができません。でも Bk から定まるエラー位置多項式の係数 Λ1... Λk とシンドロームの間には下のような線形な関係が成り立ちます。

$$\text{エラ位置多項式:} 1 + \Lambda_0 X + \Lambda_1 X^2 + \ldots + \Lambda_k X^{k+1}$$
$$\equiv (1+\beta_0 X)(1+\beta_1 X)\ldots(1+\beta_k X)$$
$$\begin{pmatrix} S_0 & S_1 & S_j \\ S_1 & S_2 & S_{j+1} \\ & & \\ S_j & S_{j+1} & S_{2j} \end{pmatrix} \begin{pmatrix} \Lambda_0 \\ \Lambda_1 \\ \\ \Lambda_j \end{pmatrix} = \begin{pmatrix} S_{j+1} \\ S_{j+2} \\ \\ S_{2j+1} \end{pmatrix}$$

このエラー位置多項式の係数はシンドロームの行列演算を使って計算できます。なお、エラー位置多項式の次数が、発生しているエラーの個数を表します。　エラー位置多項式が逆行列で求められるとしても逆行列を求めること自体が複雑な処理です。発生するエラーが二つ以下ならば二次の逆行列計算までで済ませられます。3 個以上のエラー位置計算を行う逆行列回路は複雑なものになります。でもエラー位置多項式の係数とシンドロームにはサイクリックな関係があります。これを利用してエラー位置多項式を求める Berlekamp Massey Algorithm が見つかっています。(文献 [1],[2]) このアルゴリズムを

valarray<ClGf> を使って記述すると次のようになります。

```
bool GetEPP(const valarray<GF>& crVlrGfSyndromeAg
                                , valarray<GF>& rVlrGfLamdaAg)
{
    valarray<GF> vlrGfBeforeAt(rVlrGfLamdaAg.size(),Gf(0));
    rVlrGfLamdaAg = GF(0);
    rVlrGfLamdaAg[0] = 1;
    vlrGfBeforeAt[0] = 1;

    for( int n=1, int inLAt=0, int inXAt=1, GF gfBAt=GF(1);
            n<crVlrGfSyndromeAg.size(); n++ ){
        Gf gfDAt;
        gfDAt = tfInnerProduct(
                    rVlrGfLamdaAg[slice(n-inLAt-1,inLAt+1,-1)]
                  , crVlrGfSyndromeAg[   slice(0,inLAt+1,1)]);
        if ( gfDAt == 0 ){
            inXAt++;
        }else if( 2*inLAt > n){
            // gfDAt != 0 and 2*inLAt > n
            rVlrGfLamdaAg -= gfDAt*gfBAt
                            * vlrGfBeforeAt.shift(inXAt);
            inXAt++;
        }else{
            if ( n+1-inLAt > rVlrGfLamdaAg.size() ){
                return false;
            }
            // gfDAt != 0 and 2*inLAt <= n
            valarray<GF> vlrTempAt(rVlrGfLamdaAg.size())
                    = rVlrGfLamdaAg;
            rVlrGfLamdaAg -= gfDAt*gfBAt
                            * vlrGfBeforeAt.shift(inXAt);
            inLAt = n+1 - inLAt;
            vlrGfBeforeAt = vlrTempAt;
            gfBAt = gfDAt;
            inXAt = 1;
        }
    }
    return true;
}
```

この GetEPP(.) 関数は数学記号を使って記述した Berlekamp-Massey アルゴリズムと同程度の行数です。しかもプログラム記述には、数学記号で記述したときの曖昧さが残りません。これは STL/valarry の記述力の高さを例示しています。

またこの関数は実際にパソコン上で動作します。このような STL/valarray を使ったモデル記述により、効率の良いシステム設計検討ができます。

７．ECC エンコーダーの valarray/VrfyLib 実装

「付録 1」に ECC 生成多項式が四次のときのエンコード関数 ClRs::Encode(.) を示します。そこで使う変数や関数を kVerifier に登録するためのコードも同時に示します。

ClRs::Encode(.) 関数はエンコード係数の初期値設定を除けば、ECC コードの次数に無関係に次の三行で記述できます。

```
void ClRs::Encode(const ClGf& crGfAg)
{
            ・
    ClGf gfAt = crGfAg + m_vlrGfEcc[crGfAg.size()-1];
    m_vlrGfEcc = m_vlrGfEcc.shift(-1);

    m_vlrGfEcc += vlrCoeffStt*gfAt;
    //delayWithCpu(10,k_uS);
    //clTestVctDStt.EndWait()
}
```

ClRs::Encode(.) の呼び出しと引数の設定は下の test vector から行います。main 関数から呼び出す必要はありません。このようなテスト・ベクタを coverage 100% になるように作ります。

```
+1c clRsStt.EncodeMakeSyndromBuf __function 2 encode 0 1
+1c clRsStt.EncodeMakeSyndromBuf __function 6 decode¥

                              0 1 0x0f 0x36 0x7d 0x40
+0c                                    __wait
+0c clRsStt.m_vlrGfEcc       __compare 0x28 0x14 0x0a
```

８．ECC エンコーダーの SystemC 実装

valarray<ClGf> で記述したコードは SytemC では動きません。回路合成できません。SystemC の RTL 記述に変換する必要があります。「付録 2」 に SystemC での Reed Solomon エンコード回路例を示します。

valarray<ClGf> での記述とは異なり、ECC 4 バイトのレジスタごとに明示的に Syndrom 計算を行っています。valarray<ClGf> を使った記述から SystemC への記述は特に難しくありません。プログラム この Reed Solomon Encoder のときはシーケンシャルな動きがないだけ、RTL 記述への変更が容易です。

Reed Solomon Encoder 程度の単純なコードならば valarray<ClGf> から SystemC RTL 記述への変更でバグが入り込む可能性は殆どありません。でも DVD-RW の codec のように規模が大きくなり、またソフトとハードが関係しあうようになると RTL 記述への変更に伴ってバグが入り込む可能性が高くなります。でも kVerifier は RTL 記述への変換にさもなって動作が同じであることを保証できます。記述した動作が同じならば、valarray<ClGf> の動作確認をした test vector ファイルが SystemC RTL 記述の動作確認にも使えるからです。

９．最後に

valarray と SystemC による Reed Solomon エンコーダーを実装し、kVerifier 上でシミュレーション・テストを行いました。

実際のシステム設計・検討では、インターリーブ、クロス ECC でのエラー訂正などを考慮しなければなりま

せん。ソフト、ハードの役割分担を検討し、処理速度を見積もらねばなりません。

システム設計・検討では SystemC 記述より STL と kVerfier 仕様記述スレッドの組み合わせを使ったほうが効率的です。kVerifier を使えば、ソフト・ハードの強調動作も含めてシミュレーション・テストが可能です。そのときに作成したテスト・ベクタは RTL 実装でも使えます。

ソフト、ハードの協調シミュレーション・テストを可能にする kVerifier は ICE の入手が難しい SoC で特に有効です。採用検討してくださることをお願いします。

参考文献

[1]http://www.ee.ucla.edu/~matache/rsc/slide.html
Encoding/Decoding Reed Solomon Codes
Adina Matache
Department of Electrical Engineering
University of Washington

[2]http://golay.uvic.ca/nltr/98_sep/awards/0122mass.pdf
Shift-Register Synthesis and BCH Decoding l
IEEE Trans. Inform. Theory, IT-13, pp. 21-27, 1967. JL Massey, "Shift-register
synthesis and BCH decoding,"

========== Appendix 1==========

```
#include<kcommon.h>
#include<gf.h>
#include<valarray>
#include<list>
#include<testVct.h>
const int cInRsEccLength = 4;
class ClRs{
      std::valarray<ClGf>    m_vlrGfEcc;
      std::list<ClGf>          m_lstInputData;
   protected:
   public:
      ClRs(void);
      void Encode(const ClGf& crGfAg);
      void MakeSyndrom(const ClGf& crGfAg);
      void EncodeAtAll(void);
      void MakeSyndromAtAll(void);

      // use Initialize() at data flow befoer using Encode()/MakeSyndrom()
      void Initialize(void);

#ifdef DfVrfy
      ClGf& m_rGf;       // to debug
      void EncodeMakeSyndromBuf(const std::string& crStrAg);

      void Register(kk::ClTestVct* pClAg, const std::string& crStrAg)
      {
            kk::tfRgstVerified(pClAg
               , kk::tfNewVfClFnctnSpl(
                     this, &ClRs::EncodeMakeSyndromBuf,
crStrAg+".EncodeMakeSyndromBuf"
                  )
            );
            //kk::tfRgstMntrVrfy(pClAg, m_vlrGfEcc, crStrAg+".m_vlrGfEcc");
            kk::tfRgstVfMt(pClAg
                  , kk::tfNewVfCntnr(m_vlrGfEcc, crStrAg+".m_vlrGfEcc")
                  , kk::tfNewMtCntnr(m_vlrGfEcc, crStrAg+".m_vlrGfEcc")
            );
      }
#endif   // DfVrfy
};

            .
void ClRs::Encode(const ClGf& crGfAg)
{
      static ClGf arGfCoeffStt[]={
                                    ClGf::GetAlphaI(6),

                                    ClGf::GetAlphaI(3)
                                    + ClGf::GetAlphaI(4)
                                    + ClGf::GetAlphaI(5)
                                    + ClGf::GetAlphaI(6),

                                    ClGf::GetAlphaI(1)
                                    + ClGf::GetAlphaI(2)
                                    + ClGf::GetAlphaI(4)
                                    + ClGf::GetAlphaI(5),

                                    ClGf::GetAlphaI(0) // MSB conefficient
                                    + ClGf::GetAlphaI(1)
                                    + ClGf::GetAlphaI(2)
                                    + ClGf::GetAlphaI(3)
      };
      static valarray<ClGf> vlrCoeffStt(arGfCoeffStt,4);

      ClGf gfAt = crGfAg + m_vlrGfEcc[crGfAg.size()-1];
      m_vlrGfEcc = m_vlrGfEcc.shift(-1);

      m_vlrGfEcc += vlrCoeffStt*gfAt;
}

            .
static ClRs clRsStt;
void ClRs::EncodeMakeSyndromBuf(const string& crStrAg)
{
      int inAt, inCountAt;
      string strCommandAt;
      istringstream istrmAt(crStrAg);
      istrmAt.unsetf(ios::basefield);
      istrmAt >> inCountAt;
      istrmAt >> strCommandAt;
      m_lstInputData.erase(m_lstInputData.begin(),m_lstInputData.end() );
      for(; inCountAt>0; inCountAt--){
```

```
            istrmAt >> inAt;
            m_lstInputData.push_back(ClGf(inAt));
        }
        if ( strCommandAt == "encode"){
            clRsStt.EncodeAtAll();
        }else if ( strCommandAt == "decode"){
            clRsStt.MakeSyndromAtAll();
        }else{
            assert(0);// EncodeMakeSyndromBuf(.) command parameter error
        }
}

static class ClTestVctD : public ClVrfyLibTstVct {
   public:
      ClTestVctD(void):ClVrfyLibTstVct("RS_by_STL"){}
      virtual void doAtInitial( const string& crStrAg);
} clTestVctDStt;

void ClTestVctD::doAtInitial( const string& crStrAg)
{
      if ( IsSameNocase(crStrAg,"test.vrf")
         || IsSameNocase(crStrAg,"testSTL.vrf")
      ){
            ClTestVct::doAtInitial(crStrAg);    // open crStrAg file
            cout << "Now simulation input file is " << crStrAg << " in rs.cpp" <<
endl;

            clRsStt.Register(this, "clRsStt");
      }
}
#endif    /* DfVrfy*/
```

========== Appendix 2 ==========

```
#pragma warning(disable:4786)     // 256 文字以上の warning を殺す
#include <VrfySysc.h>

#include<rs.h>

using namespace kk;

SC_MODULE(CtCodec) {
      sc_in_clk    i_clk;
      sc_out<bool> o_blErrorOut;
      sc_out<sc_uint<8> > i_data;
      sc_signal<bool>    m_sgBlTest;
      sc_signal<unsigned int> m_inArSindrom[4];
      //sc_signal< unsigned int > m_sgWdTest;
      //sc_signal< sc_uint<8> > m_sgWdTest2;

      sc_signal<sc_uint<8> > m_sgData;

      SC_CTOR(CtCodec)
      {
            SC_CTHREAD(encode, i_clk.pos()); // Clock に同期して動作する
関数を作る
            i_clk( ClVfIntfSyscSg::GetStt()->GetClock() );
            i_data(m_sgData);
            o_blErrorOut(m_sgBlTest);
      }
      void encode();

#ifdef DfVrfy
      void testFunction(const string& crStrAg)
      {
            int inAt=3;

            m_sgBlTest.write(true);
            m_sgBlTest = true;
            m_sgBlTest = (bool)inAt;

      }
      void Register(kk::ClTestVct* pClAg, const std::string& crStrAg)
      {
            tfRgstVerified(pClAg
                  , tfNewVerified( m_inArSindrom[0],
crStrAg+".m_inArSindrom_0") );
            tfRgstVerified(pClAg
                  , tfNewVerified( m_inArSindrom[1],
crStrAg+".m_inArSindrom_1") );
            tfRgstVerified(pClAg
                  , tfNewVerified( m_inArSindrom[2],
crStrAg+".m_inArSindrom_2") );
            tfRgstVerified(pClAg
                  , tfNewVerified( m_inArSindrom[3],
crStrAg+".m_inArSindrom_3") );
      }
#endif
}ctCodecStt("codec");

void CtCodec::encode()
{
      extern unsigned int rs_getCmbntnLgcMul(unsigned int wdRtAg, unsigned
int wdLtAg);
      m_inArSindrom[0].write( rs_getCmbntnLgcMul(m_inArSindrom[0],
0x40) );
      m_inArSindrom[0].write( m_inArSindrom[0].read() ^ m_sgData.read() );

      m_inArSindrom[1] = rs_getCmbntnLgcMul(m_inArSindrom[1], 0x78);
      m_inArSindrom[1].write( m_inArSindrom[1].read() ^ m_sgData.read() );

      m_inArSindrom[2] = rs_getCmbntnLgcMul(m_inArSindrom[2], 0x33);
      m_inArSindrom[2].write( m_inArSindrom[2].read() ^ m_sgData.read() );

      m_inArSindrom[3] = rs_getCmbntnLgcMul(m_inArSindrom[3], 0x0f);
      m_inArSindrom[3].write( m_inArSindrom[3].read() ^ m_sgData.read() );
}

/*---------- Beginning of kVerifier code ----------*/
#ifdef DfVrfy
//#include <VrfyLib.h>
using namespace std;
using namespace kk;

class ClTestVctD2 : public ClVrfySyscTstVct {
   public:
      ClTestVctD2(const string crStrAg):ClVrfySyscTstVct(crStrAg){}
      virtual void doAtInitial( const string& crStrAg);
};
```