暗号プロトコルの実装を支援するための アプリケーションフレームワーク

宮嵜仁志[†1]　毛利公美[†2]　白石善明[†1]

ネットワーク上で第三者に知られてはならない秘匿情報をやりとりする際に暗号プロトコルが使われる．安全性が理論の上で証明されている暗号プロトコルでも，実装上の脆弱性により必須の処理を迂回されてしまうこともある．秘匿情報を安全に活用するには，安全性の品質を持たせながら暗号プロトコルの実装を容易にできるような開発環境が提供されることが望ましい．一定以上の品質のアプリケーションを開発したいという要求を実現するためにアプリケーションフレームワークが利用される．ソフトウェアの構造を決め，処理の流れを制御するフレームワークならば，実装上の脆弱性を作ってしまう危険性を軽減できる．

本稿では暗号プロトコルの実装にモデルベース開発とソフトウェアパターンを適用したアプリケーションフレームワークを提案し，再利用性を評価する．提案したフレームワークを利用して暗号プロトコルアプリケーションを試作したところ，アプリケーションコード中のフレームワークコードが占める割合は 7 割程度であり，本フレームワークは高い再利用性を持ち，暗号プロトコルアプリケーションの開発を支援できることを確認した．

A Framework for Secure Implementing Cryptographic Protocol

HITOSHI MIYAZAKI[†1]　MASAMI MOHRI[†2]　YOSHIAKI SHIRAISHI[†1]

1. はじめに

クラウドやビッグデータなどの計算機，ネットワークの利用形態が注目されている．これらで動作するサービス，アプリケーションにおいて，個人情報や業務上の機密情報を扱うときにはセキュリティ対策が必須となる．

通常，サービスやアプリケーションを実現するトップダウン型の設計や，新しいサービスやアプリケーションを創出するための要素技術のボトムアップ型の研究開発のいずれでも，ネットワーク，計算機の利用形態に応じる形で，セキュリティ要求を抽出し，その要求を満たすように暗号プロトコルが構成される．ここでのトップダウン型の設計とは，サービスやアプリケーションを実現するシステムの設計が先にあり，要求分析をした後にすでに部品として利用可能な適当な暗号プロトコルを導入することを表す．ボトムアップ型の研究開発とは，シーズとしての暗号技術を実現し，そのシーズをもとに新しいサービスやアプリケーションを開発することを表す．トップダウン型とボトムアップ型のいずれにしても，最終的にはシステムを実現するときの構成要素となる暗号プロトコルが実装コードの形でシステムに組み込まれることになる．

暗号プロトコルは暗号演算の機能だけでなく，プロトコルのシーケンス制御機能やデータ形式の変換など，暗号プロトコルの理論的な機能モジュール以外にも実装では必要となる機能モジュールがあるといった，理論と実装の間にはこのような隔たりがある．このことは安全性が理論の上で証明されている暗号プロトコルでも実装上の脆弱性により必須の処理をバイパスされてしまうことの懸念[1]と関係する．今後もネットワーク上で秘匿情報を扱うサービス，アプリケーションが増加していくことが予想される中では，秘匿情報を適切に取り扱うために，安全性の品質を持たせながら高度な暗号プロトコルの実装を容易にするような開発環境の実現とその提供は，システムの要素技術である暗号プロトコルの開発とともに重要な課題である．

これまでに暗号プロトコルの実装を支援するための暗号演算を始めとしたライブラリが開発，提供されている．しかし，暗号演算以外の機能モジュールはサポートされていないものも多い．実装の品質は安全性に直結するが，シーケンス制御機能のように他の機能と複雑に絡む機能で，ソフトウェアの広範囲の設計に影響を及ぼすような，ライブラリだけで支援できないものもサポートされていない機能モジュールとしてあげられる．実装を容易にする以外の目的として，一定以上の品質のアプリケーションを開発したいという要求を実現するためにもしばしばアプリケーションフレームワークが利用される．ソフトウェアの設計を決め，処理の流れを制御するフレームワークならば，ライブラリだけでは支援できない機能の実装を支援できる．暗号プロトコルの実装に適切な設計がなされたフレームワークを利用することで，実装上の脆弱性を作ってしまう危険性を軽減する効果があると考えられる．

本稿では暗号プロトコルの実装を支援するためのアプリケーションフレームワークを提案する．フレームワークを実装して，それを用いていくつかの暗号プロトコルを試

†1 名古屋工業大学
Nagoya Institute of Technology
†2 岐阜大学
Gifu University

作する．アプリケーションコード中のフレームワークコードのステップ数の比率を測定し，フレームワークが高い再利用性を持ち，暗号プロトコルアプリケーションの開発を支援できることを確認する．

2. アプリケーションフレームワーク

2.1 フレームワークの定義

ソフトウェアの開発において，一度作成した設計やモジュールを別の開発の際に再利用することで開発効率と品質を高めるという取り組みがなされてきた．フレームワークはそうした取り組みの中で生まれたオブジェクト指向に基づいたソフトウェアの再利用技術の一つである．

フレームワークには多様な定義がある．フレームワークの定義としてもっとも多く使われるのは「抽象クラスの集合とそのインスタンス間の相互作用によって表現された，システムの全体または一部の最利用可能な設計」である．他によく使われる定義としては「開発者がカスタマイズできるアプリケーションの骨組み」というものがある[2]．これら 2 つはそれぞれフレームワークの構造と利用目的という異なる点について着目した定義であり，相反するものではない．

本稿ではフレームワークを構造と再利用性の観点から捉える．フレームワークの適用先のアプリケーションの共通機能を抽出し，フレームワークの利用者が抽出した共通機能の組み換えや新たな機能の追加を容易に行えるような構造の雛形をフレームワークとする．

2.2 フレームワークを利用したアプリケーションの開発

アプリケーション開発者が記述するコードをユーザコードと呼ぶ．Pree はフレームワークの構成には適用先によってコードの変更がないフローズンスポットと呼ばれる箇所と，適用先のアプリケーションに合わせてアプリケーション開発者が追加，変更をするホットスポットと呼ばれる箇所があると述べている[3]．フレームワークを利用した開発では，アプリケーション開発者はフレームワークのフローズンスポットには手を加えず，作成したアプリケーションに合わせてホットスポットにユーザコードを記述してアプリケーションコードを作成する．

アプリケーション開発者はフレームワークを利用することによって，自身が記述するユーザコードが減りコーディングが容易になる．また，アプリケーション開発においてコーディングよりも設計作業の方が時間のかかる工程とされている．フレームワークを利用したアプリケーション開発においては，アプリケーションの設計の一部がフレームワークによって規定されるため，設計にかかる時間を短縮できる．設計はアプリケーションの品質を決める重要な要素であるため，フレームワークを利用することによって一定の品質以上のアプリケーションを作成できるという利点もある．フレームワークを利用したアプリケーション開発は，実装の容易性とアプリケーションの品質の面で有効である．

3. 対話処理と状態遷移モデル

3.1 対話処理

プログラムの処理形態は大きく 2 つに分けられる．一つはユーザがプログラムの実行前にジョブを一括して依頼し，ジョブに含まれる情報のみにしたがって処理を行うバッチ処理である．もう一方は，プログラムの実行中に入力が必要になったときに，ユーザが入力を行いプログラムの実行を制御する対話処理と呼ばれる処理形態である[4]．対話処理の代表的なものとしてはグラフィカルユーザーインターフェース（GUI）アプリケーションが挙げられる．また，ネットワークなどを介して複数のコンピュータ間で行われる通信プロトコルにしたがったデータの送受信も広義の対話処理に分類される．

3.2 状態遷移モデル

ソフトウェア開発をするときに品質を高める方法の一つとしてモデルベース開発と呼ばれる手法がある．モデルベース開発とは設計の初期段階で，実現したい機能をモデルで表現し，後の設計と開発でソフトウェアがモデルのとおりに動作することを検証しながら進めていく開発手法である．モデルを用いることで，ソースコードを書く前に開発するソフトウェアの動作をある程度確認することができるため，開発設計段階でソフトウェアの仕様や要求に問題がないかを早期に確認できる利点がある[5]．他にソフトウェアを設計段階で抽象化するため，作成したソフトウェアのソースコードの再利用率が高くなりやすい利点もある．

ソフトウェア開発で用いられるモデリング手法の一つに状態遷移モデルがある．状態遷移モデルはリアクティブシステムを記述する表現方法の一つである．リアクティブシステムは，外部から何らかの入力を与えるとそれに対応した出力を返す，ということを繰り返し実行するシステムである．状態遷移モデルを用いた開発では，すべての状態に対してすべての起こりうる入力を与えたときの出力を定義した状態遷移関数を記述する．こうすることで，開発段階で入力に対しての出力の定義に漏れがあり，システムが予期しない動作をする，停止するといった事態を防ぐことができる．以上のことからモデルベース開発はソフトウェアの品質と再利用性を高められる開発手法であり，中でも状態遷移モデルは高い信頼性を要求するソフトウェアの開発に有効であるといえる．

3.3 暗号プロトコルの状態遷移モデルでの記述

暗号プロトコルは相手からの入力に対して暗号演算を行い，プロトコルの参加者に情報を出力する，リアクティブシステムの一種であるとみなすことができる．暗号プロトコルはシーケンス図で記述できる．暗号プロトコルを表現したシーケンス図は状態の有限集合を暗号演算のアルゴ

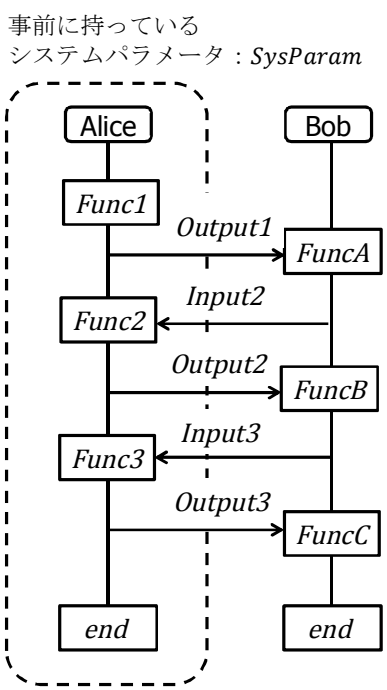

図 1　暗号プロトコルの一般的な流れ

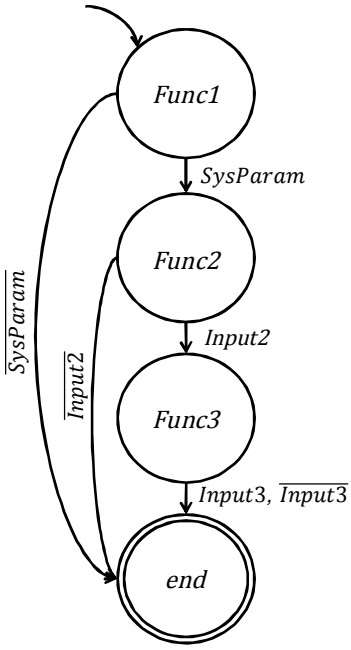

図 2　Alice の振る舞いを表現した状態遷移図

リズムとした決定性有限オートマトン（Deterministic Finite Automaton: DFA）に変換できる．

例えば，図１に示す暗号プロトコルの一般的な流れを表現したシーケンス図中の Alice の振る舞いは，図２に示す DFA に変換できる．図２に示す DFA であるM_{Alice}は状態の有限集合をS，文字の有限集合をΣ，遷移関数を（T:$S \times \Sigma \rightarrow S$），開始状態を$s \in S$，受容状態の集合を$A \subseteq S$として次のように表現される．

$M_{Alice} = (S, \Sigma, T, s, A)$であるとき，

$S = \{Func1, Func2, Func3, end\}$

$\Sigma = \{SysParam, \overline{SysParam}, Input2, \overline{Input2}, Input3, \overline{Input3}\}$

$s = \{Func1\}$

$A = \{end\}$

$T = \{$

$Func1 \times SysParam \rightarrow Func2,$

$Func1 \times \overline{SysParam} \rightarrow end,$

$Func2 \times Input2 \rightarrow Func3, \quad Func2 \times \overline{Input2} \rightarrow end,$

$Func3 \times Input3 \rightarrow end, \quad Func3 \times \overline{Input3} \rightarrow end$

$\}$

以下，M_{Alice}の動作を順に説明する．

1. Alice の初期状態は$Func1$である．
2. 状態$Func1$に正しいシステムパラメータ$SysParam$が入力されたら一回目の暗号演算$Func1$を実行して状態$Func2$に遷移する．誤ったシステムパラメータ$\overline{SysParam}$が入力された場合は受容状態endに遷移する．
3. 状態$Func2$に正しい入力$Input2$が与えられたら，二回目の暗号演算$Func2$を実行して状態$Func3$に遷移する．誤った入力$\overline{Input2}$が与えられた場合は受容状態endに遷移する．
4. 状態$Func3$に正しい入力$Input3$，誤った入力$\overline{Input3}$のどちらが与えられた場合でも三回目の暗号演算$Func3$を実行して，受容状態endに遷移する．

4. ソフトウェア設計のためのパターン

4.1 パターンとは

一般にフレームワークには，複数の適用場面においてフレームワークコードの再利用性が高いこと，フレームワークを適用する際に機能の追加，削除，修正といった変更が容易であることが求められる．これらの要求を満たすにはフレームワークを適用する対象のソフトウェアの共通機能の抽出，各機能の分離を適切にしなければならない．フレームワークの適切な設計をするための方法として，ソフトウェアを実装する上での制約条件と問題，解決方法の組を抽象化したパターンの適用が有効である．パターンは３つのカテゴリーに分類される．抽象度の高い順に，ソフトウェア全体の構造を表現するアーキテクチャパターン，ソフトウェア中のコンポーネント間の関係を表現するデザインパターン，言語などのプラットフォーム固有の実装手法であるイディオムがある[6]．本稿では暗号プロトコルアプリケーションの処理形態に合わせたアーキテクチャパターン，状態遷移モデルの開発に適したデザインパターンを適用することで設計品質を高めている．なお，イディオムについては言語依存のコーディングの技法であり，フレームワークの性質に関わるものではないため本稿では言及しない．

4.2 MVC パターン

対話型処理のアプリケーションに適したアーキテクチ

```
public class TransState
    public static void main() {
        int nextFunc = 1;
        while(nextFunc<4)
            switch(nextFunc) {
                case 1:
                    (Func1の処理);
                    nextFunc++;
                    break;
                case 2:
                    (Func2の処理);
                    nextFunc++;
                    break;
                case 3:
                    (Func3の処理);
                    nextFunc++;
                    break;
                default:
                    break;
            }
        }
    }
}
```

図 3　switch-case 文による状態遷移コード

```
public class TransState
    public static void main() {
        State func = new Func1();
        while(func!=null) {
            func = func.execute();
        }
    }
}

public class Func1 extends State {
    public State execute() {
        （Func1の処理）;
        return new Func2();
    }
}

public class Func2 extends State {
    public State execute() {
        （Func2の処理）;
        return new Func3();
    }
}

public class Func3 extends State {
    public State execute() {
        （Func3の処理）;
        return null;
    }
}
```

図 4　State パターンを適用した状態遷移コード

ャパターンとして Model-View-Controller（MVC）パターンがある．MVC パターンは，GUI アプリケーションや Web アプリケーションの開発などでよく用いられるパターンである．MVC パターンを適用すると，アプリケーションの機能を処理（Model），出力（View），入力と処理の依頼（Controller）の 3 つのコンポーネントに分離することによって，各機能の独立性を高めることができる．３つのうちどれか一つのコンポーネントを変更した場合でも，他のコンポーネントへの影響を抑えられる．

4.3 State パターン

状態遷移モデルの設計に適したデザインパターンとして一つの状態を一つのクラスで表現する State パターンがある．State パターンの適用によってソースコードのメインの処理を変更することなく，任意の DFA を記述できる．例えば図 2 のような$Func1$，$Func2$，$Func3$の順に状態が遷移するプログラムを簡略化して記述すると，State パターンを適用しなかった場合のコードは図 3，State パターンを適用した場合のコードは図 4 のようになる．State パターンを適用していないコードに$Func1$と$Func2$の間に新たに$FuncA$を追加しようとすると，$Func2$と$Func3$をそれぞれ case3 と case4 へ 1 つずつ後ろにずらすといったように，追加したいコード以外のコードの変更量が大きい．State パターンを適用したコードに同様の操作をするには，新たに State を継承したクラス$FuncA$を実装し，$Func1$から呼び出すクラスを$FuncA$に変更し，$FuncA$から次のクラス$Func2$を呼び出せばよい．

5. フレームワークの設計

5.1 設計方針

本稿で提案するフレームワークの対象の適用先は暗号プロトコルアプリケーションとする．再利用性と拡張性を備えたフレームワークを設計するには，フローズンスポットとホットスポットを適切に設定しなければならない．フローズンスポットとホットスポットを切り分けるには，各機能の分離がなされていなければならない．そこでフレームワークの設計に 4 章で述べた 2 種類のパターンを適用する．暗号プロトコルアプリケーションのモジュール構成を図 5 に示す．図 5 において斜体で表記されているモジュールは本稿で提案するフレームワークでは共通機能として抽出しないホットスポットである．

暗号プロトコルアプリケーションは，二者以上が互いに要求と応答を繰り返しながら処理を進めていく対話処理である．暗号プロトコルアプリケーションに，対話型処理に適したアーキテクチャパターンである MVC パターンを適用すると，構成機能を暗号演算機能（Model），通信出力機能（View），通信入力機能（Controller）の 3 つに大きく分割できる．

暗号演算機能は，暗号プロトコルの種類によって利用する暗号方式が異なるため，一般的な暗号プロトコルの範囲で共通機能として抽出するものではない．また，暗号演算機能の実装は既存のライブラリで支援されている．提案フレームワークでは暗号演算機能そのものは実装せず，フレ

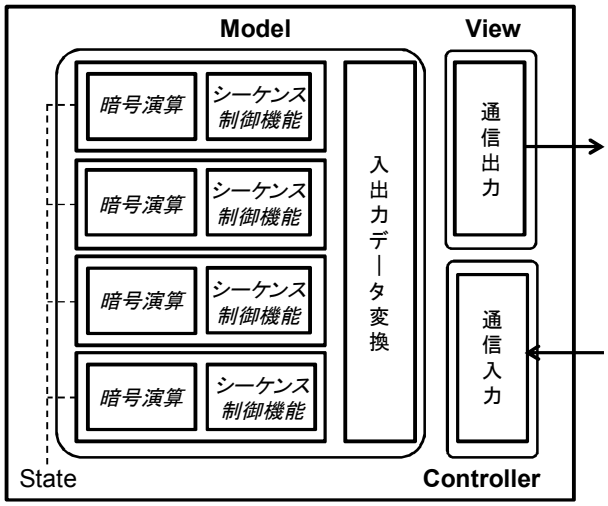

図 5　暗号プロトコルアプリケーションの機能モジュール

ームワークを利用する開発者が任意の暗号演算機能を追加できるホットスポットを提供する．

通信入出力機能は，機能を階層化して考えると，暗号プロトコルとは別の層で動作する．つまり，暗号プロトコルは特定の通信プロトコルに依存しない．一般的な暗号プロトコルの範囲で共通機能として抽出できるので，提案フレームワークでは通信機能をフローズンスポットとして提供する．

理論的な暗号プロトコルの機能モジュール以外で，実装に必要な機能モジュールにシーケンス制御機能がある．シーケンス制御機能は暗号プロトコルの処理のタイミングと順序を決定する，暗号プロトコルの実装で安全性に直結する特に重要な機能である．この機能が正しく実装されないと，必須の処理が迂回される，プロトコルが途中で停止するなどの予期しない動作が起きるため，フレームワークを利用する暗号プロトコルアプリケーション開発者が変更しないフローズンスポットとしたい．しかし，プロトコル中の処理の順序は暗号プロトコル間で異なる．シーケンス制御機能は，すべては共通機能として抽出できないため，一部をホットスポットとする．

3.2 節で暗号プロトコルアプリケーションの振る舞いを暗号演算のアルゴリズムを一つの状態とした状態遷移モデルで記述できることを述べた．暗号プロトコルアプリケーションに，状態遷移モデルに適した State パターンを適用することで，アプリケーションの流れのすべてを把握することなく，各暗号プロトコル固有の振る舞いを記述できる．また，一つの状態に一回の暗号処理を独立して記述するため，実装した暗号処理のコードの再利用性を高めることができる．上述の理由からシーケンス制御機能には State パターンを適用する．各状態には暗号演算とシーケンス制御機能の 2 つのホットスポットの組が含まれる．2 つのホットスポットの組を載せた State パターンの状態を作成するための雛形をフローズンスポットとして提供する．状態の組み換えを容易にするために，状態の入力と出力の形式を同一のものとした．通信の入出力をする際にそれぞれで利用する個別の形式に変換をする．

暗号プロトコルフレームワークの全体像は，暗号演算を一つの状態とした組み換えが容易な状態遷移モデルに，通信用のインターフェースを付加したものになる．

5.2 フレームワークの構成

暗号プロトコルのフレームワークの構成を図 6 に示す．5.1 節で述べたようにアーキテクチャパターンとして MVC パターン，デザインパターンとして State パターンを適用した．MVC パターンの適用でホットスポットとして設定した暗号演算部を変更しても，フローズンスポットとして設定した通信の入出力へ影響を及ぼさないようにした．State パターンの適用によって，状態並び替え，追加，削除などの変更を，4.3 節で述べたように状態の内部だけを編集すれば操作できるようにしている．設計方針では触れていないが，暗号プロトコルアプリケーションに必要な機能として State パターンの現在の状態から次の状態へのデー

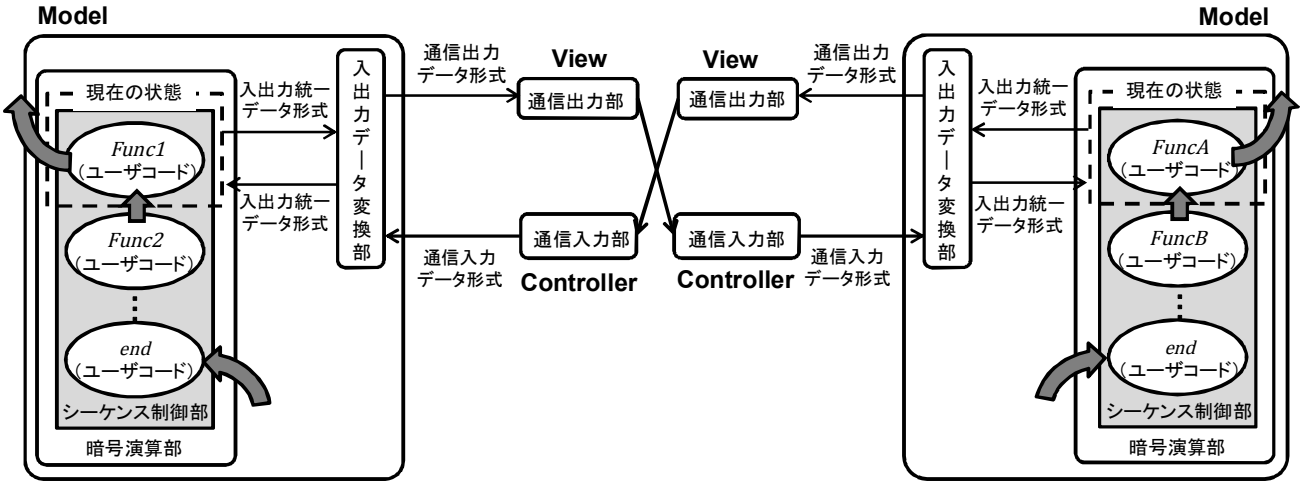

図 6　暗号プロトコルフレームワークの構成

表 1　フレームワークの開発環境

OS	Windows7 Professional SP1 64bit
言語	Java（JDK1.7.0_09）
Web Server	Apache Tomcat 7.0.34[7]
標準以外のライブラリ	AjaxBaron[8] Gson[9] Apache Commons Codec[10]

```
public class Func1 extends State { //Stateクラスの継承
    public Func1(PreData pd) { //コンストラクタ
        setPreData(pd); //一つ前の状態からの引き継ぎデータ
    }

    /**
     * 状態Func1での処理を実行する
     * @param IOBean 通信相手からの入力
     * @return 次の状態
     */
    @Override
    public State execute(IOBean in) {
        Predata pd = getPreData(); //引き継ぎデータを取得

        /*暗号演算を記述する*/
        String str = RSA.enc(in.getParam1(), "publicKey")

        setURL("http://xxx.jp/yyy/zzz"); //出力先URLを指定
        setOut(new IOBean(str)); //演算の出力結果をセット
        return new Func2(new PreData("publicKey"));
    }
}
```

図 7　ホットスポットのサンプルコード

タの受け渡し機能を追加した．

フレームワークの動作を説明する．通信入力部に入力されたデータは入出力データ変換部によって入出力統一データ形式に変換される．変換されたデータは暗号演算部にある最初の現在の状態 *Func1* に入力され，演算処理が実行される．*Func1* の処理が終わったら演算処理の結果が状態から出力されたのち，状態が遷移して *Func2* が現在の状態になる．演算処理の結果は入出力統一データ形式で出力される．このデータは入出力データ変換部によって通信出力データ形式へ変換され，通信出力部から通信相手に出力される．通信相手も同様の手順で *FuncA*，*FuncB* が順に現在の状態となり，状態内の処理が実行される．この動作はそれぞれ自身の状態が受容状態 *end* になるまで繰り返される．

6. 実装

6.1 実装したフレームワーク

5 章で述べた設計に従ってフレームワークを実装した．開発言語には Java を使用した．実装したフレームワークはクライアント・サーバーモデルで動作するものである．フレームワークの開発環境を表 1 に示す．サーバ側のアプリケーションは Apache Tomcat[7]を利用して Web サーバとして実装した．エンティティ間の通信機能は他の Web サービスとの連携を考え HTTP プロトコルとした．また，通信するデータ形式はクライアントの実装がブラウザ上で動く JavaScript などでも容易となるよう URL クエリパラメータと JavaScript Object Notation（JSON）形式とした．フレームワークの実装には Java の標準ライブラリ以外にフリーウェアの AjaxBaron[8], Google 社が提供する Gson[9], Apache Commons Codec[10]を使用した．

6.2 暗号プロトコルの試作

Apache Tomcat は広く使われている認証，暗号通信用の機能である基本認証，ダイジェスト認証，SSL 通信などを Web サーバの標準機能として提供している．提案フレームワークを用いて，ダイジェスト認証，クライアントの公開鍵認証，SSL 通信と同様の機能を持ったアプリケーションを試作した．共通鍵暗号方式には AES 暗号，公開鍵暗号方式には RSA 暗号，ハッシュ関数には SHA-1 を利用した．

提案フレームワークのホットスポットを拡張して暗号プロトコルアプリケーションを実装する際のコード例を図 7 に示す．フレームワークを利用する開発者は抽象クラス State を継承して実装をする．メソッド execute 内に暗号演算などの処理を記述する．メソッド execute 内から前の状態からの引き継ぎデータにアクセスし，return 文で次の状態と引き継ぐデータを指定できる．また，処理の出力結果をセットし，出力先を選択できる．

7. 評価

6.2 節で述べた試作した 3 つの暗号プロトコルアプリケーションについて，アプリケーションコード全体の中のフレームワークコードが占めるステップ数の割合を測定した．測定結果を表 2 に示す．暗号プロトコルアプリケーションを試作する際に，自作の通信，暗号化などのライブラリを利用したが，フレームワークの有効性を評価するため測定結果の合計のステップ数にライブラリのステップ数は加えていない．ライブラリのステップ数を除いた合計ステップ数を 100%としたとき，フレームワークコードが占める割合はクライアントアプリケーションとサーバーアプリケーションの合計で，ダイジェスト認証では 84%，公開鍵によるクライアントの認証では 75%，SSL 通信では 66%であった．すなわちアプリケーションコードのうち，それぞれ 84%，75%，66%が再利用可能なコードで構成できているということである．通信，暗号演算の回数が多くなるほどユーザコードのステップ数が増えるため，相対してフレームワークコードが占める割合が減る傾向にあるが，アプリケーションコードの再利用率は全体として 7 割程度と高い値を示した．この理由としては，提案したフレームワークでは通信機能などのインターフェース機能をフローズンスポットとしただけでなく，暗号プロトコルの振る舞いを状態遷移モデルで表現し，任意の DFA を容易に構成できるような部

表 2　アプリケーションコード全体の中のフレームワークコードが占めるステップ数の割合

		ダイジェスト認証	クライアントの公開鍵認証	SSL通信
クライアント	フレームワークコード	205	205	205
	アプリケーションコード全体	244	284	302
サーバ	フレームワークコード	203	203	203
	アプリケーションコード全体	244	261	314
アプリケーションコード中に フレームワークコードが占める割合		84%	75%	66%

品を作成し，粒度の細かい部分までフローズンスポットとできたことが挙げられる．また，フレームワークを利用して暗号プロトコルアプリケーションを実装する際に State パターンの状態部分のみを変更すればよいようにした．ホットスポットを集約したことでオブジェクト間のデータの受け渡しなどのオブジェクト指向特有のオーバーヘッドとなるコードを削減できたことも理由として考えられる．

8. まとめ

本稿では暗号プロトコルの実装を支援するためのアプリケーションフレームワークを提案した．提案したフレームワークは状態遷移モデルを用いて開発することで，入力に対しての出力の定義漏れを防ぐことができる設計となっている．アーキテクチャパターンとして MVC パターンを，デザインパターンとして State パターンを用いることで，フレームワークの再利用性を高め，アプリケーションの処理の流れのすべてを把握することなく，容易に各暗号プロトコル固有の振る舞いを記述できる．実装した提案フレームワークを利用してダイジェスト認証，クライアントの公開鍵認証，SSL 通信と同様の機能を持ったアプリケーションを試作し，フレームワークを評価した．ライブラリを除いた暗号プロトコルアプリケーションのステップ数のうち，フレームワークコードが占める割合は全体として 7 割程度であった．フレームワークが高い再利用率を持ち，フレームワークの利用により暗号プロトコルアプリケーションの開発を支援できることを確認した．

本稿で提案したフレームワークのセッション管理機能は Apache Tomcat の機能を利用している．三者以上でセッションを共有できる機能があれば，提案フレームワークを三者以上の任意の暗号プロトコルにも適用可能なものへ拡張できると考えられる．これを今後の検討課題としたい．

参考文献

1) 独立行政法人情報処理推進機構：自動車の情報セキュリティへの取り組みガイド，(2012)．
2) Ralph E Johnson，中村宏明，中山裕子，吉田和樹：パターンとフレームワーク，共立出版（1999）．
3) W.Pree（著），佐藤啓太，金澤典子（訳）：パターンとフレームワーク，共立出版（1999）．
4) 松尾啓志：オペレーティングシステム，森北出版（2005）．
5) 渡辺政彦：組込みソフトウェア向け開発支援環境，情報処理，Vol.45，No.1，pp.10-15（2004）．
6) F.Buschmann, R.Meunier, H.Rohnert, P.Sommerlad, M.Stal（著），金澤典子，水野貴之，桜井麻里，関富登志，千葉寛之（訳）：ソフトウェアアーキテクチャ ソフトウェア開発のためのパターン体系，近代科学社（2000）．
7) Apache Tomcat: Apache Tomccat(online), available from<http://tomcat.apache.org/>(accessed 2013-05-16).
8) T.Misawa: Ajax Baron(online), available from<http://web2driver.com/ajax/index.php?Ajax%20Baron>(accessed 2013-05-16).
9) Google: google-gson(online), available from<http://code.google.com/p/google-gson/>(accessed 2013-05-16).
10) Apache Commons: Commons codec(online), available from<http://commons.apache.org/codec/>(accessed 2013-05-16).