Vol.2011-MBL-57 No.9
Vol.2011-UBI-29 No.9
2011/3/7

通信プログラム開発を支援する データ入出力および検証コード生成

坂根裕[†]　鈴木敦志[†]　岡田昌也[††]

言語非依存のデータフォーマット定義から、高品質かつ汎用性の高いデータ入出力および検証コードを自動生成する技術について報告する．本稿では，複雑な構造を持つデータフォーマットを，特定パターンの組み合わせで表現する記述方式を開発した．解析手順が単純でないビットマップを例に，構造記述と自動生成を試みた．その結果，より少ない工数で目的のコードが得られることが分かった．

Code Generation of Data Format I/O and Validation Programs for Network Software Developments

YUTAKA SAKANE[†]　ATSUSHI SUZUKI[†]
MASAYA OKADA[††]

We have developed a data description language for existing non-standardized data formats in order to generate data I/O and validation program source codes. By describing Device Independent Bitmap (DIB/BMP) data format which have many variants, we have shown that the code generator can provide general-purpose, quality-guaranteed data handling program with much less effort.

1. はじめに

通信機能を備える情報機器やシステム開発では，高品質（不具合がなく安定稼動し，保守が容易）な通信機能の実現に，開発・テスト工数を多く必要とする．近年では，ウェブ連携サービスが注目され，REST[1]や WCF(Windows Communication Foundation)[2]など，開発を支援する各種技術は整ってきた．しかし，特定の業務システム開発などでは，新技術が適用できない既存のデータフォーマットやプロトコルのサポートが必須要件となることも多い．このような領域では未だ，低レベル API を用いた通信機能の開発が求められる．

筆者らは，特殊な構造を持つデータフォーマットやプロトコルを，いくつかのパターンの組み合わせとして表現し，データの入出力と検証コード（プログラム）を自動生成する技術開発を進めている．提案手法では，(1)特定の開発言語に依存せず自由なコード出力ができること，(2)組込機器なども想定し使用リソース量を調整できること，(3)通信機能以外の開発の見通しを良くするため，システム本体と生成コードを疎結合にすることを方針としている．

本稿では，バイナリベースのプロトコルを対象に，データ構造記述言語の設計と開発を行った．開発した言語でビットマップ等の既存のデータフォーマットを定義し，コード生成を行った結果について報告する．

2. 通信機能の開発

2.1 通信機能開発における問題意識

高品質な通信機能の開発には，ソフトウェアだけではなく処理を実行するハードウェアやネットワークなど，幅広く深い知識が必要となる．通信プログラム開発を困難なものにしている要因として，以下の理由が挙げられる．

1. **非同期処理を考慮しなくてはいけない**.

データ送受信には時間がかかるため，システムのメインスレッドとは非同期で処理させることが多い．そのため，受信データ処理やユーザインタフェースの更新タイミングなどに気を配る必要がある．

2. **ネットワークに関するエラー検出と対処が複雑になる**.

ネットワークや通信先の状態を事前に知ることはできず，通信結果から推測するしかない．例えば，特定サーバへのアクセスに応答がない場合，ネットワークの問題でデータが不到達なのか，サーバの問題で応答が不能なのかを切り分けることは難しい．

[†] デジタルセンセーション株式会社
Digital Sensation Co., Ltd.
[††] 静岡大学
Shizuoka University

Vol.2011-MBL-57 No.9
Vol.2011-UBI-29 No.9
2011/3/7

3. 処理速度やメモリ量などを考慮しなくてはいけない.

ネットワーク経由のデータアクセスでは，基本的にシーケンシャルアクセスが前提となる．そのため，メモリ上に展開できる小さなデータであれば問題ないが，動画のような巨大なデータを取り扱う場合には，受信したデータをどのように扱うか（メモリに保持，ローカルストレージに保存，破棄するなど）も考慮しなくてはいけない．

図 1 に，外部サーバのストレージに存在する巨大データを受信し，データに処理を施して外部ストレージへ再送信する例を 2 種類示す．図上段では，データを一旦ローカルストレージに保存し，「受信」「処理」「送信」の処理を順番に行っている．図下段では，データを受信しながら解析し，必要なデータが抽出できた時点で処理，送信している．

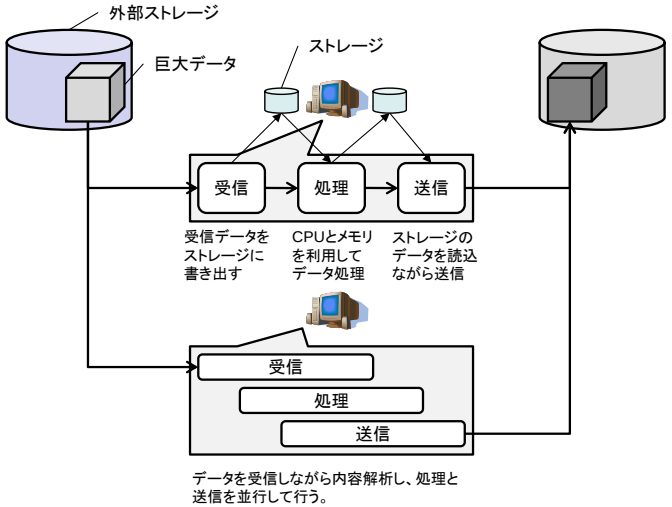

図 1　通信機能の実装シナリオ
Figure 1　Implementation scenarios of communication functions.

どちらの例も同じデータ構造を用いているが，上段はプログラムが処理毎に分離しているためコードの見通しがよく保守も容易である．一方，処理速度や使用リソース量の観点から見れば下段の例の方が効率は良い．このことは，業務に係わる（画像処理などの）データ処理以外に，通信に関する（データ送受信・解析タイミング制御などの）データ処理が存在することを意味する．業務に関するデータ処理と通信に関するデータ処理を可能な限り疎結合に（分離）することで，開発者は業務に関するデータ処理に専念できる．

2.2 提案手法

本研究では，データ構造を定義するための「言語」を開発者に提供する．開発者は対象となるデータやプロトコルを本言語で定義し，出力言語（プログラミング言語）を選択してコード生成することを想定している．出力言語を自由に選択させることで，プログラムの実行対象となる環境に特化したコードが生成でき，実行効率の最適化及び開発効率の向上が狙える．

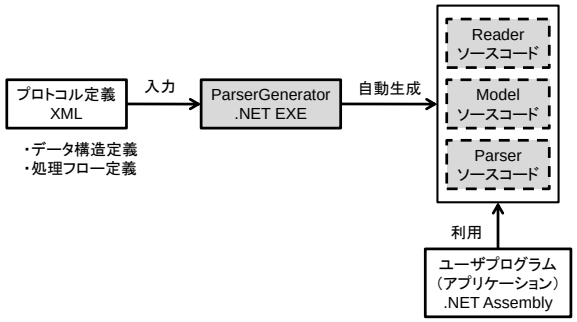

図 2　データ構造定義からのコード生成
Figure 2　Code generation from data structure definitions.

図 2 に，データ構造定義とコード生成の関係を示す．現時点では，データ構造定義は XAML(Extensible Application Markup Language)で記述する（最終的には特化言語を設計する）．記述したデータを ParserGenerator に入力すると，Reader, Model, Parser が生成できる（現状では Writer は生成されない）．また，生成するコードのプログラミング言語は C#のみとなっている．開発者は，開発中のシステムへ生成されたコードを組み込み利用する（現状では.NET Assembly のみが対象となる）．

ParserGenerator は，データ解析のために抽象度の異なる 2 種類の API をプログラムコードとして生成する．Reader は軽量なデータ解析器であり，PULL 型の API を提供し，以下の特徴を有する．

- データモデルを構築せず，一度の操作で単一のフィールドのデータを提供する．
- 任意の時点でデータ解析が中断できる．
- データ内容の解釈を行わないため，情報の欠落なしに全てのデータを取得できる．
- 動作が軽量であり，入力データ量に係わらず常に一定量のメモリを消費する．
- データエラー等のイベント時に，現在のストリーム位置と解析コンテキストを扱いやすい形で出力できる．
- API の利用方法はやや煩雑であり，利用には対象のデータ構造に対する知識が必要となる．

Vol.2011-MBL-57 No.9
Vol.2011-UBI-29 No.9
2011/3/7

Parser は Reader を利用し，データモデルの構築とユーザプログラムのイベント駆動を行うイベント駆動型パーサであり，以下の特徴を有する．

- ある粒度のモデルが構築できるまでデータの解析を進め，解析完了をイベントとして通知する．
- 任意の時点でデータ解析を中断できる．
- 消費するメモリ量は，処理粒度（サイズ）により異なる．

Reader のデータ解析方法は，事前にデータブロック（小さなデータ構造）を定義しておき，ストリームデータの先頭から順にデータブロックを当てはめるように解析を進める（当てはまるデータブロックが存在しない場合は，解析エラーかデータブロックが未定義の場合）．図 3 に，ストリームデータにデータブロックを当てはめる例を示す．図左が対象となるデータ（上部がデータの先頭），図右側が事前定義したデータブロックである．図の例では，対象データは先頭に「○○ヘッダ」，続いて「△○データ」「□○データ」「○×データ」と解析が進む．

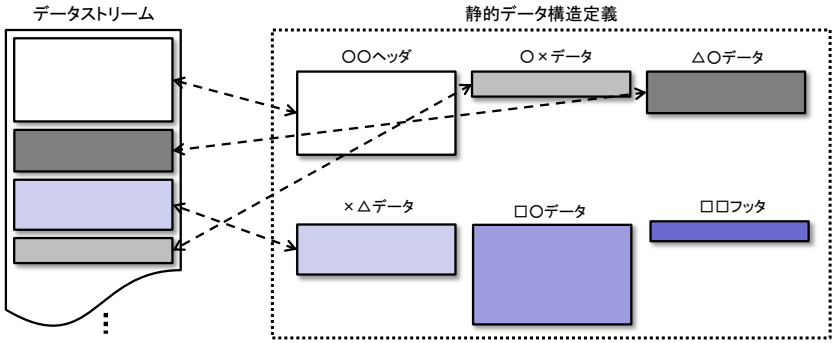

図 3　データの解析方法
Figure 3　Data parsing method.

データブロック自体の構造は，C 言語の構造体に似ており，中は複数のフィールドで構成される．フィールドは 4 種類存在し，一般的なプログラミング言語で利用する変数型である primitive 型，サイズが動的に指定されるバイト配列型，特定のデータブロック，データブロックの配列となる．

具体的で少し複雑な例として，ビットマップファイルのデータ構造を図 4 に示す．ビットマップは，ファイルヘッダ，情報ヘッダ，画像データのブロックとして分割できる．ファイルヘッダは，図に示す通り primitive 型のフィールドのみで構成される固定長のデータブロックになる．

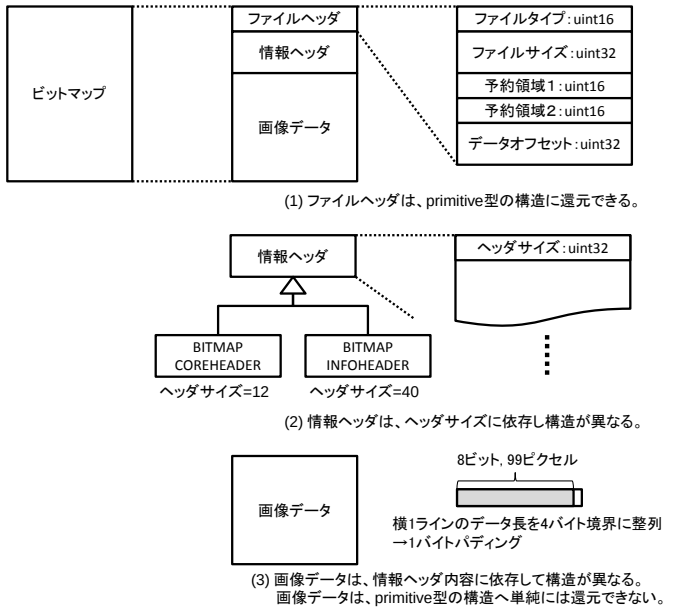

図 4　ビットマップの構造例
Figure 4　Data structure sample of the bitmap file.

図中央の情報ヘッダは複数のバージョンが存在する．図では 2 種類の例を示しているが，情報ヘッダの最初のフィールド（ヘッダサイズ）の値で，どのデータブロックか決定できる．さらに，情報ヘッダ内に記載される色数によりパレット情報を含むかどうかも決まる．最後に画像データは，情報ヘッダに記載された深度の色情報，サイズを持つ配列と考えることができるが，行単位で 4 バイト境界に整列するようパディングが行われる．

このように特殊な構造を持つデータフォーマットは，処理速度，可読性，拡張性など，各々異なる設計思想に基づき設計されており，統一的に記述するためには以下の要件全てに対応する必要がある．

- バイナリベース，テキストベース（XML など），ハイブリッド（HTTP など）
- バイトオーダー（リトルエンディアン／ビッグエンディアン／可変）
- 文字コード（ASCII，ISO2022 系，UNICODE 系など）
- N バイト境界へのフィールド配置
- 可変長フィールドの終端方法（バイト長指定，要素数指定，デリミタ，改行，正

Vol.2011-MBL-57 No.9
Vol.2011-UBI-29 No.9
2011/3/7

規表現，EOF）
- データのチャンク化，圧縮
- 階層的プロトコル（DICOM over DICOM Upper Layer，SOAP over HTTP など）

本研究では，上記要件のうち，先ずはバイナリベースのプロトコルに使われる要件をサポートする．

データ内容の検証は，フィールド単位で行う．ジェネレータで生成されるパーサは，どのフィールドを解析しているのかの状態を管理し，1 ステップずつ読み進めながら問題があるとイベントにより通知する．

3. 実験

構造定義言語の表現力と生成されるコード内容を検証するために，実際にビットマップの構造を記述した．ビットマップの種類は，文献[3]に掲載される全 7 種類を対応させた．記述した XAML コードと生成された C#のコードの一部を図 5 と図 6 に示す．図 5 左の XAML で，PrimitiveFieldDefinition 要素として Image Width を定義している．生成されたコードを見ると，Image Width の読み込みが成功すると Image Height 状態へ遷移するように記述されており，1 ステップずつ解析するコードとなっている．

また，記述言語の要素の説明を表 1 に纏める．ビットマップの情報ヘッダのように，途中まで解析が進まないとブロックの型が決定しないようなケースも，図 6 のように構造体の継承関係として記述することでコード生成できる．

実験で記述した XAML のコード量は約 400 行で，開発時間は 2 時間程度であった．生成された C#のコード量は約 2200 行（約 1000 ステップ）となった．

```
<ConcreteStructDefinition
    Name="BitmapCoreHeader"
    BaseType="DIB Header">
    <ConcreteStructDefinition.Body>
        <PrimitiveFieldDefinition
            Name="Image Width"
            Type="UInt16"/>
        <PrimitiveFieldDefinition
            Name="Image Height"
            Type="UInt16"/>
        <PrimitiveFieldDefinition
            Name="Planes"
            Type="UInt16"/>
        <PrimitiveFieldDefinition
            Name="Bits per Pixel"
            Type="UInt16"/>
    </ConcreteStructDefinition.Body>
</ConcreteStructDefinition>
```

```
private global::System.Boolean ExecuteBitmapInfoHeader_ImageWidth_PrimitiveValue()
{
    global::System.Boolean succeeded = base.ExecuteInt32PrimitiveValue(
        "Image Width", base.DefaultByteOrder, false);
    if (!succeeded)
    {
        this.Current = State.Interrupted;
        return false;
    }
    base.Next = State.BitmapInfoHeader_ImageHeight_PrimitiveValue;
    return true;
}
```

図 5　構造定義の記述例と生成されたプログラム（1）
Figure 5　A description example of the data definition language and a generated code (1).

表 1　データ構造記述言語の要素
Table 1　Elements of the data definition language.

要素/属性名		説明
DataStructureMetaModel		**データ構造定義のトップレベル要素**
属性	Name	データ構造定義の名前
属性	Namespace	生成されるコードの名前空間
要素	Global	データ全体の配置を定義するFieldDefinition
要素	Structs	StructDefinitionの配列
属性	DefaultByteOrder	各数値型の既定のバイトオーダーを指定する
属性	DefaultEncoding	文字列型の既定のエンコーディングを指定する
AbstractStructDefinition		**抽象構造体(複合型)を定義する要素**
属性	Name	構造体の名前
属性	BaseType	構造体の継承元の型の名前
要素	Header	ヘッダフィールドの定義
要素	Trailer	トレイラーフィールドの定義
ConcreteStructDefinition		具象構造体(複合型)を定義する要素
属性	Name	構造体の名前
属性	BaseType	構造体の継承元の型の名前
要素	Body	ボディフィールドの定義
PrimitiveFieldDefinition		**単純型(整数等)フィールドを定義する要素**
属性	Name	フィールドの名前
属性	Type	フィールドの型名
属性	Length	(Optional) フィールドの長さ(バイト)
ArrayFieldDefinition		**可変長のバイト列フィールドを定義する要素**
属性	Name	フィールドの名前
属性	Type	フィールドの型名
属性	Delimitation	フィールドの終端方法
StructFieldDefinition		**構造体を参照するフィールドを定義する要素**
属性	Name	フィールドの名前
属性	Type	フィールドの型の構造体の名前
SequenceFieldDefinition		**繰り返し参照されるフィールドを定義する要素**
属性	Name	フィールドの名前
属性	ItemType	繰り返される構造体の型の名前
属性	Delimitation	フィールドの終端方法

Vol.2011-MBL-57 No.9
Vol.2011-UBI-29 No.9
2011/3/7

```
<AbstractStructDefinition
    Name="DIB Header">
    <AbstractStructDefinition.Header>
        <PrimitiveFieldDefinition
            Name="DIB Header Size"
            Type="UInt32"
            PreserveContext="True"/>
    </AbstractStructDefinition.Header>
</AbstractStructDefinition>
<ConcreteStructDefinition
    Name="BitmapCoreHeader"
    BaseType="DIB Header">
    <ConcreteStructDefinition.Body>
        <PrimitiveFieldDefinition
            Name="Image Width"
            Type="UInt16"/>
```

```
        private global::System.Boolean ExecuteDibHeader_DibHeaderSize_PrimitiveValue()
        {
            global::System.Boolean succeeded = base.ExecuteUInt32PrimitiveValue("DIB
Header Size", base.DefaultByteOrder, true);
            if (!succeeded)
            {
                this.Current = State.Interrupted;
                return false;
            }
            global::DigitalSensation.DataFormat.StructFieldContext ctx =
base.ContextStack.Peek() as global::DigitalSensation.DataFormat.StructFieldContext;
            if (this.CanSpecializeToBitmapCoreHeader(base.Buffer, ctx))
            {
                ctx.Type = "BitmapCoreHeader";
                base.Next = State.BitmapCoreHeader_ImageWidth_PrimitiveValue;
                return true;
            }
            if (this.CanSpecializeToBitmapCoreHeader2(base.Buffer, ctx))
            {
                ctx.Type = "BitmapCoreHeader2";
                base.Next = State.BitmapCoreHeader2_ImageWidth_PrimitiveValue;
                return true;
            }
            if (this.CanSpecializeToBitmapInfoHeader(base.Buffer, ctx))
            {
                ctx.Type = "BitmapInfoHeader";
                base.Next = State.BitmapInfoHeader_ImageWidth_PrimitiveValue;
                return true;
            }
            if (this.CanSpecializeToBitmapInfoHeader2(base.Buffer, ctx))
            {
                ctx.Type = "BitmapInfoHeader2";
                base.Next = State.BitmapInfoHeader2_ImageWidth_PrimitiveValue;
                return true;
            }
            if (this.CanSpecializeToBitmapV3Header(base.Buffer, ctx))
            {
                ctx.Type = "BitmapV3Header";
                base.Next = State.BitmapV3Header_ImageWidth_PrimitiveValue;
                return true;
            }
            if (this.CanSpecializeToBitmapV4Header(base.Buffer, ctx))
            {
                ctx.Type = "BitmapV4Header";
                base.Next = State.BitmapV4Header_ImageWidth_PrimitiveValue;
                return true;
            }
            if (this.CanSpecializeToBitmapV5Header(base.Buffer, ctx))
            {
                ctx.Type = "BitmapV5Header";
                base.Next = State.BitmapV5Header_ImageWidth_PrimitiveValue;
                return true;
            }
            this.OnError(
                ErrorSeverity.Error,
                global::System.String.Format(
                    "unable to find appropriate concrete type for {0}",
                    ctx.Type));
            base.Next = State.Interrupted;
            return false;
        }
```

図 6　構造定義の記述例と生成されたプログラム（2）
Figure 6　A description example of the data definition language and a generated code (2).

4. 関連技術・研究

データ構造定義からの入出力コード生成に関する取り組みとして，Protocol Buffers[4]，Thrift[5]等の実装が利用されている．これらの定義言語は，分散システム環境下でのメッセージ交換手順を隠蔽するための新しいデータ構造定義を目的としており，既存のデータ構造定義を目的の一つとする本手法とは異なる．

ネットワークソフトウェアのデータ入出力形式の定義言語として，Binpac[6]，Prolac[7]，Preccs[8]，April++[9]等の研究が存在する．これらは，既存のネットワークプロトコルのデータ構造表現が可能であるが，ネットワークソフトウェアに固有のストリームやメッセージ交換等の概念を前提としている．

本手法と同じように，ネットワークソフトウェア開発に限らない汎用的なデータ構造の定義言語として，PADS[10]，DATASCRIPT[11]等が存在する．本提案手法では，オブジェクト指向言語のクラス定義に類似した馴染みある定義と，さまざまな実行環境を想定し,データ解析をフィールド単位で詳細に制御できる柔軟さを実現している．

5. おわりに

正しいデータを正しく解析するプログラムは比較的簡単に構築できるが，不正なデータを正しく不正と判断し（可能であれば）そのまま解析が続行でき，メインプログラムを停止させない機能の開発は困難である．扱うデータが同じでも，ソフトウェアが稼働する環境や制約が異なるとゼロからの開発になるケースも多く，ノウハウ蓄積は「再利用が困難なコードのみ」という開発現場は多いと思われる．

本研究では，新しいプロトコルの設計だけではなく，既存のデータフォーマットやプロトコルをも対象とし，開発言語や実行環境に極力依存しない通信プログラム（データ入出力と検証機能）の自動生成を試みた．通信や解析に関するデータ処理をメインプログラムから分離して記述できると，全体構成の見通しがよく，保守も容易な開発が可能となることが分かった．

今後，Writer の生成機能や出力コードの品質を高める研究を進める．また，生成したコードに対する品質の測定方法やテスト方法についても検討を進め，開発現場で実用可能なレベルにしたい．

謝辞　本稿は，総務省・戦略的情報通信研究開発推進制度(SCOPE)地域 ICT 振興型研究開発「地域医療連携における異種医療機器間の「しなやかな」メッセージ交換ツールの研究開発(102306012)」の支援を受けて行った．ここに記して謝意を表す．

Vol.2011-MBL-57 No.9
Vol.2011-UBI-29 No.9
2011/3/7

参考文献

1) Roy Thomas Fielding: “Architectural Styles and the Design of Network-based Software Architectures,” PhD Thesis, University of California at Irvine, Incormation and Computer Science (2000).
2) Microsoft, "Windows Communication Foundation,” http://msdn.microsoft.com/ja-jp/netframework/aa663324 (2011 年 2 月 1 日 確認).
3) Wikipedia: “BMP file format,” http://en.wikipedia.org/w/index.php?title=BMP_file_format&oldid=409094366 (2011 年 2 月 1 日 確認).
4) Google: “Protocol Buffers,” http://code.google.com/intl/ja/apis/protocolbuffers/ (2011 年 2 月 1 日 確認).
5) Apache Software Foundation: “Thrift,” http://thrift.apache.org/ (2011 年 2 月 1 日 確認).
6) Ruoming Pang, Vern Paxson, Robin Sommer, Larry Peterson: “binpac: a yacc for writing application protocol parsers,” In Proceedings of the 6th ACM SIGCOMM conference on Internet Measurement, pp.289-300 (2006).
7) Eddie Kohler, M. Frans Kaashoek, David R. Montgomery: “A readable TCP in the Prolac protocol language,” In Proceedings of the conference on applications, technologies, architectures, and protcols for computer communication, pp.3-13 (1999).
8) 服部健太, 平木敬: “プロトコル記述言語 Preccs における入出力コードの生成方式,” 電子情報通信学会技術研究報告. CPSY, コンピュータシステム Vol.106 No.436, pp.19-24 (2006).
9) 阿部勝幸, 岩崎英哉, 河野健二: “アプリケーション層プロトコルの記述に基づく拡張性に優れたプロトコル処理コード生成系,” 日本ソフトウェア科学会 第 22 回大会論文集 (2005).
10) Kathleen Fisher, Robert Gruber: “PADS: a domain-specific language for processing ad hoc data,” In Proceedings of the 2005 ACM SIGPLAN conference on Programming Language design and implementation (PLDI), pp.295-304 (2005).
11) Godmar Back: “Datascript - a specification and scripting language for binary data,” In Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (DPCE), pp.66-77 (2002).