VDM-SL 仕様からの Smalltalk プログラムの自動生成

小田 朋宏
株式会社 SRA
tomohiro@sra.co.jp

荒木 啓二郎
九州大学
araki@csce.kyushu-u.ac.jp

要旨

VDM-SL は実行可能なサブセットを持ち，複数のインタプリタ実装が提供されている形式仕様記述言語である．本稿は，*Smalltalk* 環境上に構築された *ViennaTalk* ライブラリに実装された，*VDM-SL* による実行可能仕様からの *Smalltalk* プログラムの自動生成器の実装を説明し，生成された *Smalltalk* ソースコードの性能評価を示す．

1. はじめに

ソフトウェアシステムの仕様を形式的に記述することで 仕様記述の問題点をより早期に発見し修正する開発手法として, 形式手法が注目されている [1, 2]．形式仕様記述言語 VDM-SL [3] は実行可能なサブセットを持ち，複数のインタプリタ実装が提供されている [4]．実行可能な VDM 仕様は vdmUnit 等のテストフレームワークを利用して単体テストを継続的に行うことで，高い品質の仕様記述が可能であることが事例から知られている [5].

筆者らはこれまで VDM-SL インタプリタを Smalltalk 環境から利用するためのクラスライブラリ ViennaTalk を開発し，仕様の妥当性を確認するための UI プロトタイピング環境 Lively Walk-Through [6], や Web API プロトタイプ環境 Webly Walk-Through[6, 7] および，WebIDE サーバ VDMPad [8, 9] を構築してきた．ViennaTalk は Smalltalk 環境の柔軟性を活かして，VDM-SL で記述された実行可能仕様を Smalltalk 環境の一部として利用するための Smalltalk ライブラリである．ViennaTalk は 既存の VDM-SL インタプリタを外部インタプリタとして利用し，Smalltalk プログラムの動作の一部として VDM-SL 仕様を実行する仕組みを提供することで，VDM-SL のための開発ツールを開発することを容易にする．

本稿では、ViennaTalk に実装した，VDM-SL 仕様から Smalltalk プログラムを自動生成する機能について述べる．VDM-SL 仕様から Smalltalk プログラムを自動生成し，他の Smalltalk プログラムと連携して動作させることで，インタプリタとの連携に起因するオーバーヘッドを減らし，インタプリタ実行よりも実行効率のよい連携を可能にした．

本稿では，VDM-SL の概要を紹介し，自動生成の概要，特に VDM の特徴でもある強力な集合演算，写像，およびパターンマッチの実現について説明する．生成されたプログラムの性能評価として，エラトステネスの篩のアルゴリズムを列を多用した記述，集合を多用した記述，および写像を多用した記述を作成し，それぞれについて処理時間を VDMTools [10] および VDMJ [11] のインタプリタ実装と比較した．VDMTools および VDMJ はともに実際のソフトウェア開発で利用された実績のあるインタプリタ実装である．

2. VDM-SL の概要

VDM-SL はモデル規範型の形式仕様記述言語で，記述対象を抽象するための型，定数，関数を定義し，状態空間の定義と状態を変更する操作を定義することで，システムの入出力や状態変化を記述する．インタプリタによって実行可能なサブセットを持つが，プログラミング言語とは異なり，仕様記述は動作させることが目的ではなく，システムの動作における入出力や状態変化を規定することを目的として記述する．また，インタプリタは実行すること自体が目的ではなく，記述されたシステムの動作をシミュレートすることでその振る舞いを理解し確認することが目的である．

VDM-SL にはモジュール機能があるが，本稿では簡単のためモジュールを持たない記述について説明す

る．VDM-SL による仕様記述は types で始まる型定義部，values で始まる定数定義部，functions で始まる関数定義部，state で始まる状態定義部，および operations で始まる操作定義部からなる．各定義部は同種の定義部が複数存在しても良く，また，必要がなければ省略しても良い．

VDM-SL で提供されている型には，自然数型，非 0 自然数型，整数型，実数型，文字型，ブール型，引用型，トークン型などの基本型と，複合型，列型，集合型，直積型，合併型，オプション型，写像型などの合成型がある．文字列のための型はなく，文字列は文字型の列 seq of char として扱う．

2.1. エラトステネスの篩と双子素数

エラトステネスの篩は素数列を求める古典的なアルゴリズムである．2 から N までの間にある素数を求める場合，まずは素数候補となる 2 から N までの自然数の昇順の列を定義する．そして，候補列から先頭要素を取り出した数を素数として素数の列に加え，その全ての倍数を候補列から削除する操作を，候補列が空になるまで続けることで，素数列を得ることができる．双子素数とは，隣り合う 2 つの奇数がともに素数となっているペアである．例えば，3 と 5，5 と 7，11 と 13 は双子素数である．

図 1 に，素数を求めるエラトステネスの篩と，双子素数を求めるアルゴリズムを VDM-SL で記述した実行可能な仕様を示す．この仕様には型定義部および定数定義部はなく，状態定義部，関数定義部，および 2 つの操作定義部から成っている．

図 1 の状態定義部では，候補列 space および 素数列 primes の 2 つの変数が非 0 自然数の列として定義されている．init 句では，初期値として共に空列を定義している．

続いて operations で始まる操作定義部では，setup, next, sieve の 3 つの操作が定義されている．setup 操作は非 0 自然数を引数として取り，候補列 space に 2 から引数の値までの自然数の昇順の列，素数列 primes に空列 [] をそれぞれ代入する．next 操作は候補列 space が空かどうかを確認し，空の場合には 終了をあらわすために nil を返し，空でない場合には先頭要素 hd space を素数列 primes の最後尾に連結し，篩にかける．篩をかける sieve 操作は，列の内包表記を使って候補列 space から引数で与えられた数

```
state Eratosthenes of
    space : seq of nat1
    primes : seq of nat1
init s == s = mk_Eratosthenes([], [])
end
operations
    setup : nat1 ==> ()
    setup(x) ==
        (space := [i | i in set {2, ..., x}];
        primes := []);

    next : () ==> [nat1]
    next() ==
        if
            space = []
        then
            return nil
        else
            (dcl x:nat1 := hd space;
            space := tl space;
            primes := primes ^ [x];
            sieve(x);
            return x);

    sieve : nat1 ==> ()
    sieve(x) ==
        space :=
            [space(i) |
            i in set inds space &
            space(i) mod x <> 0];
functions
    twins : seq of nat1 -> seq of (nat1* nat1)
    twins([x1, x2] ^ rest) ==
        let ts =
            if rest = []
            then []
            else twins([x2] ^ rest)
        in
            if x2 - x1 = 2
            then [mk_(x1, x2)] ^ ts
            else ts;
operations
    prime10000 : () ==> seq of nat1
    prime10000() ==
        (setup(10000);
        while next() <> nil do skip;
        return primes);

    twinprime10000 : () ==> seq of (nat1* nat1)
    twinprime10000() ==
        (setup(10000);
        while next() <> nil do skip;
        return twins(primes));
```

図 1. エラトステネスの篩の VDM-SL 仕様

で割った剰余が 0 でない，すなわち引数の倍数でない要素のみからなる列を作成し，それを新たな候補列 space としている．

functions で始まる関数定義部では，非 0 自然数列から隣り合う要素同士の差が 2 となるペアを列にして取り出す関数 twins がパターンマッチを使って定義されている．パターンマッチは関数 twins の仮引数 [x1, x2] ^ rest で使われている．これは，2 要素の列にマッチし第 1 要素を x，第 2 要素を y とする列の列挙パターンと，それと任意の列にマッチし rest とするパターンを連結した列連結パターンであり，rest には列の列挙パターンの残り部分である第 3 要素以降の部分列が束縛される．

最後に 2 番目の操作定義部では，それぞれ 10000 以下の素数列と双子素数列を求める操作 prime10000 と twinprime10000 が定義されている．

3. 自動生成

筆者らは Smalltalk プログラムと VDM-SL 仕様の連携を重視して ViennaTalk を開発してきた．コード生成においても，Smalltalk らしい表現をしたプログラムを生成すること，生成元の VDM-SL 仕様と文面上の対応関係がわかりやすいことを目標として機能設計し実装した．そのために既存のクラスやメソッドに適合するように，また，生成された Smalltalk プログラムの中で使われているメッセージ名から元の VDM-SL 仕様での記述が類推しやすいように，Smalltalk 基本ライブラリを拡張した．

本節では，VDM-SL と Smalltalk の間の型システムの違いを埋めるための対応関係，VDM-SL の組み込み関数のための Smalltalk ライブラリへの拡張，パターンマッチングや不変条件のような Smalltalk にはない言語機能の実現について説明する．

3.1. VDM-SL の型および値と Smalltalk オブジェクトの対応関係

VDM-SL と Smalltalk では型システムが大きく異なる．Smalltalk はクラスベースの動的型付き言語であり，VDM-SL を含む静的型付き言語での型に相当する言語要素としてクラスが挙げられる．VDM-SL では 1 つの値が同時に複数の型の値であり得るが，Smalltalk では 1 つのオブジェクトは 1 つのクラスに属するため，VDM-SL の型を Smalltalk のクラスに対応付けることは適切ではないと考えた．ViennaTalk でのコード生成では VDM-SL の型をクラスではなく，型を表現するオブジェクトに対応付けることとし，型を表現するためのクラスライブラリを実装した．ViennaTalk では VDM の型もオブジェクトであり，型オブジェクトにメッセージを送ることで新たな型オブジェクトを合成したり，また，型の間の関係や型と値の関係についての問い合わせを記述できる．

VDM-SL の値を Smalltalk で実現するにあたって，ViennaTalk では可能なかぎり Smalltalk の既存のクラスを利用した．VDM-SL の型システムでは，整数型の値は全て実数型の値でもある点が Smalltalk を含む多くのプログラミング言語と異なる．すなわち，値 1 は Smaltalk では SmallInteger クラスのインスタンスだが，VDM-SL では 1 は nat, nat1, int, real の 4 つの型全ての値である．ViennaTalk では SmallInteger クラスと VDM-SL の nat 型を直接対応付けるのではなく，1 というインスタンスを nat 型の Smalltalk 側の実装である ViennaNatType クラスのインスタンスに対応付ける．ViennaNatType クラスについては次節で説明する．

VDM-SL の型システムの特徴としては，型にはその値が満たすべき不変条件を付与することができる点が挙げられる．例えば，types nonzero = int inv x == x <> 0 と定義することで，非 0 整数の型 nonzero を定義することができる．

VDMTools での VDM-SL からの C++コード生成では，VDM-SL の型に対応するクラスを定義している．ViennaTalk では Smalltalk の既存のクラスを利用することで，Smalltalk での整数オブジェクトはそのまま VDM-SL での整数値として使え，また，VDM-SL での集合はそのまま Smalltalk でも Smalltalk の集合オブジェクトとして使うことができる．

VDM-SL の型と，その型を表現する Smalltalk 表現式およびそのクラス，その型の値のクラスの対応関係を表 1 に示す．

3.2. Smalltalk ライブラリの拡張

VDM-SL の特徴としては，集合や写像など抽象度の高い型と強力な組み込み機能が挙げられる．Smalltalk は豊かなクラスライブラリを持っており，集合や写像を扱う抽象度も VDM-SL と大きな違いはないが，いくつかの機能を既存のクラスに追加する必要があった．コード

表 1. VDM-SL の型とその値に対応する Smalltalk での表現式とクラス

VDM の型	VDM 表現	Smalltalk 表現	型のクラス	値のクラス
自然数	nat	ViennaType nat	ViennaNatType	Integer
非 0 自然数	nat1	ViennaType nat1	ViennaNat1Type	Integer
整数	int	ViennaType int	ViennaIntType	Integer
実数	real	ViennaType real	ViennaRealType	Float
ブール型	bool	ViennaType bool	ViennaBoolType	Boolean
引用型	<quote>	ViennaType quote: *#quote*	ViennaQuoteType	Symbol
オプション型	[*t*]	*t* optional	ViennaOptionType	*t* のクラスまたは UndefinedObject
直積型	*t1* * *t2*	*t1* * *t2*	ViennaProductType	Array
合併型	*t1* | *t2*	*t1* | *t2*	ViennaUnionType	*t1* または *t2* のクラス
集合型	set of *t*	*t* set	ViennaSetType	Set
列型	seq of *t*	*t* seq	ViennaSeqType	OrderedCollection
非空列型	seq1 of *t*	*t* seq1	ViennaSeq1Type	OrderedCollection
写像型	map *t1* to *t2*	*t1* mapTo: *t2*	ViennaMapType	Dictionary
単射型	inmap *t1* to *t2*	*t1* inmapTo: *t2*	ViennaInmapType	Dictionary
部分関数型	*t1* -> *t2*	*t1* -> *t2*	ViennaPartialFunctionType	BlockClosure
全関数型	*t1* +> *t2*	*t1* +> *t2*	ViennaTotalFunctionType	BlockClosure
複合型	compose *t* of *f1* : *t1* *f2* :- *t2* *t3* end	ViennaType compose: '*t*' of: {*f1* . false . *t1*. {*f2* . true . *t2*}. {nil . false . *t3*}}	ViennaCompositeType	ViennaComposite
不変条件	*t* inv *pattern* ==*expr*	*t* inv: [:*v* | *expr*]	ViennaConstrainedType	*t* のクラス

表 2. 既存クラスに対するコード生成のための主な拡張

既存クラス名	メソッド名	機能
Object	viennaString	対応する VDM-SL 表現式を得る
Context	viennaReturn: *value*	関数や操作のコンテキストから return する
Collection	onlyOneSatisfy: *block* power powerDo: *block*	*block* の評価結果が真になる要素を 1 つだけ持つかを判定する 冪集合を得る 全ての部分集合を列挙してそれぞれを引数として *block* を評価する
SequenceableCollection, Dictionary, BlockClosure	applyTo: *value*	*value* 番目の要素を得る
Dictionary, BlockClosure	comp: *map-or-func* ** *value*	関数合成を得る *value* 回繰り返し関数合成を行った結果を得る

表 3. コード生成のための新規クラス

追加クラス	機能
ViennaType 及び サブクラス	VDM-SL の型
ViennaComposite	複合型の値
ViennaToken	トークン型の値
ViennaComposition	関数合成の結果
ViennaIteration	繰り返し関数合成の結果

生成のための既存クラスへの拡張を図 2 に示す．この拡張により，VDM-SL が定義する全ての組み込み関数について，Smalltalk オブジェクトが提供する機能として実現された．

第 3.1 節で説明した通り，VDM-SL の型を表すオブジェクトを新規にクラスとして定義した．また，複合型およびトークン型は Smalltalk の標準クラスライブラリに直接対応するクラスがなかったために，それぞれの型の値を表現するためのクラスを定義した．関数合成や繰り返し関数合成については，既存のクラスの拡張として一般化した定義が困難であるため，合成結果を表現するクラスを定義した．表 3 にまとめて示す．

3.3. パターンマッチング

VDM-SL では強力なパターンマッチメカニズムが提供されている．パターンマッチは関数定義，λ式，集合内包表記等，仮パラメータを扱う多くの場面で利用することができる．例えば，同じ部分列が 2 度繰り返す列のみを引数として受け付け，その部分列を返す関数を定義する VDM-SL のλ式を以下に示す．

```
(lambda x^x:seq of char & x)
```

上記のλ式で，^は連結パターンと呼ばれ，マッチ対象となる列に対するあらゆる分割の中からそれぞれの部分列が 2 つのパターンにマッチする分割を得る．x ^ x は，引数として与えられた列に対して，2 つの部分列に分割してそれぞれを同一パターン x にマッチさせることで，同じ部分列の繰り返しに対してマッチする．上記のλ式に [1,2,3,1,2,3] を引数として関数適用すると [1,2,3] が得られる．また，[1,2,1,2] を引数として関数適用すると，[1,2] が得られる．同様に集合の分割に対してマッチさせる和集合パターンや，写像の分割に対してマッチさせる和写像パターンがある．

ViennaTalk では，自動生成された Smalltalk のランタイム支援を行う ViennaUtil クラスに各種パターンマッチをさせる機能を実装し，上記のλ式からは以下の Smalltalk プログラムが生成される．

```
[ :_lmd |
| x |
(((ViennaRuntimeUtil
  matchTuple:
    {(ViennaRuntimeUtil
      match: (ViennaRuntimeUtil matchIdentifier: 'x')
      conc: (ViennaRuntimeUtil matchIdentifier: 'x'))})
  value: {_lmd})
  ifEmpty: [ false ]
  ifNotEmpty: [ :binds |
    x ← binds first at: 'x'.
    true ])
  ifFalse: [ ViennaNoMatch signal ].
x ]
```

自動生成されたプログラムでは，パターンマッチの処理を行う部分が長くなり，見通しが悪くなっている．クロージャは VDM-SL のλ式には出ていない仮引数 _lmd を宣言している．そして渡されたオブジェクトに対して，ViennaRuntimeUtil を利用してネストしたパターンマッチ処理を行っている．マッチに成功した場合にはマッチした束縛環境 binds から x を取り出し，一時変数 x に代入しλ式の本体部分である x を評価する．マッチに失敗した場合には，ViennaNoMatch 例外を投げる．

上記の処理のために，パターンマッチを含む VDM-SL 仕様から生成されたプログラムは Smalltalk プログラムとしての可読性に問題がある．実際のコード生成では下記の VDM-SL のλ式での x のように識別子のみからなる単純なパターンについては，生成されたプログラムの性能や Smalltalk プログラムらしさの点から，パターンマッチを介さずにクロージャの引数として扱うなどをしている．

```
(lambda x:int & x*2)
```

生成された Smalltalk プログラムを以下に示す．

```
[ :x | x * 2 ]
```

マッチング処理を介さずに直接クロージャの引数として生成される．VDM-SL のλ式との対応関係を容易に把握することができる．

3.4. 不変条件，事前条件，事後条件，メジャー関数

VDM-SL では型および状態に対して不変条件，関数および操作に対して事前条件および事後条件，さらに再帰

関数に対して再帰の停止性の確認を支援するためのメジャー関数を定義することができる．型に対する不変条件は第 3.1 節で説明した通り，ViennaTalk では型オブジェクトに対して inv: *block* により定義することができる．

状態の不変条件についても，Smalltalk クラスを生成する場合には，生成されたプログラムにおいても不変条件が実行時検査される．状態の不変状態は，生成された Smalltalk クラスでは，inv メソッドとして定義される．Smalltalk の Slot と呼ばれる機能を利用して，VDM-SL の操作から生成された Smalltalk メソッドをコンパイルする際に，代入後に inv メソッドを実行する命令を Smalltalk メソッドのバイトコードに埋め込む．

関数および操作の事前条件および事後条件，および再帰関数のメジャー関数については，プログラムコードの生成と実行という目的においては必要ないと考え，実装していない．コード生成の目的はあくまでプログラムコードの実行であり，仕様そのものの健全性は生成されたコードの実行時の検査ではなく，仕様記述からの証明責務の生成およびインタプリタ実行によって確認されるべきである．

型の不変条件については，実行時の振る舞いに影響を与えるためにコード生成に実装した．具体的には，ある値がある型に属するかどうかを検査する VDM-SL の言語機能として is 式がある．この is 式の振る舞いを正しくコード生成に反映させるためには，型に付けられた不変条件が必要である．また，集合や列，写像の内包表現や forall 等の量化式，let-be 式や let-be 文などの型束縛による値の生成においても，仕様と一致した動作をするプログラムを生成するためには型につけられた不変条件が必要である．

4. 評価

ViennaTalk によって生成されたプログラムの性能評価として，第 2.1 節で示したエラトステネスの篩による双子素数を求める VDM-SL 仕様から生成されたプログラムの実行時間を計測した．比較対象としては，代表的なインタプリタ実装である VDMTools と VDMJ を選んだ．

また，図 1 に示した仕様記述では主に列型を使っているが，VDM-SL の仕様で列型と同様に頻繁に利用される集合型および写像型を使ったエラトステネスの篩の仕様を記述し，Smalltalk プログラムを生成して実行時間を計測した．VDM インタプリタおよび Smalltalk 環境の実行環境は以下の通りである．

```
CPU     Intel Core i5 2.5 GHz
主記憶  16GB 1600MHz DDR3
OS      Mac OS X 10.11.3 (El Capitan)
```

表 4. 列型を利用したエラトステネスの篩と双子素数の計算時間 (単位は ms)

	VT	VDMJ	VDMTools	ST
prime10000	683	4,143	24,569	40
twinprime10000	750	4,475	24,991	40

VT は ViennaTalk による生成
ST は人手で改善した Smalltalk プログラム

4.1. 列操作

図 1 の VDM-SL 仕様から生成された Smalltalk プログラムを図 2 に示す．パターンマッチ部分を除くと通常の Smalltalk プログラムとして理解可能なプログラムが生成されている．ただし，人が書いた Smalltalk プログラムと比べると，コレクションオブジェクトの不要なコピーなどがあり，人手による改善の余地が残っている．生成された Smalltalk プログラムを人手で改善したプログラムを図 3 に示す．

改善されたプログラムは，生成されたプログラムが利用しているクラスおよび制御構造に対して，より Smalltalk プログラムとして適切であると考えられるクラスおよび制御構造を使って実装された．例えば，生成されたプログラム（図 2）では space は OrderedCollection クラスのオブジェクトが使われているのに対して，人手で改良されたプログラム（図 3）では Array クラスのオブジェクトが使われており，より高速な処理となっている．

また，双子素数を発見する関数 twins については，パターンマッチではなく，要素を 1 つずつずらした列を同時に列挙してそれぞれの列からの値を処理することで，高速な処理を実現している．

ViennaTalk で生成された Smalltalk プログラム，VDMJ によるインタプリタ実行，VDMTools によるインタプリタ実行および人手で改善した Smalltalk プログ

```
| Eratosthenes next primes init_Eratosthenes
 space twins sieve setup twinprime10000
  prime10000 |
twins ← [ :_func |
| x1 x2 rest |
(((ViennaRuntimeUtil
  matchTuple:
    {(ViennaRuntimeUtil
      match:
        (ViennaRuntimeUtil
          matchSequenceEnumeration:
            {(ViennaRuntimeUtil matchIdentifier: 'x1').
            (ViennaRuntimeUtil matchIdentifier: 'x2')})
      conc: (ViennaRuntimeUtil matchIdentifier: 'rest')
      left: 2)}) value: {_func})
  ifEmpty: [ false ]
  ifNotEmpty: [ :binds |
    x1 ← binds first at: 'x1'.
    x2 ← binds first at: 'x2'.
    rest ← binds first at: 'rest'.
    true ])
  ifFalse: [ ViennaNoMatch signal ].
[ | ts |
ts ← rest = {} asOrderedCollection
  ifTrue: [ {} asOrderedCollection ]
  ifFalse: [ twins
    applyTo: {({x2} asOrderedCollection , rest)} ].
x2 - x1 = 2
  ifTrue: [ {{x1. x2}} asOrderedCollection , ts ]
  ifFalse: [ ts ] ] value ].
setup ← [ :x |
[ space ← ((2 to: x) collect: [ :i | i ])
    asOrderedCollection.
primes ← {} asOrderedCollection ] value ].
next ← [
space = {} asOrderedCollection
  ifTrue: [ thisContext viennaReturn: nil ]
  ifFalse: [
    [ | x |
    x ← space first.
    space ← space tail.
    primes ← primes , {x} asOrderedCollection.
    sieve valueWithArguments: {x}.
    thisContext viennaReturn: x ] value ] ].
sieve ← [ :x |
space ← ((1 to: space size)
  select: [ :i | (space applyTo: {i}) \\ x ~= 0 ]
  thenCollect: [ :i | space applyTo: {i} ])
  asOrderedCollection ].
prime10000 ← [
[ setup valueWithArguments: {10000}.
[ (next applyTo: {}) ~= nil ] whileTrue: [  ].
thisContext
  viennaReturn: primes) ] value ].
twinprime10000 ← [
[ setup valueWithArguments: {10000}.
[ (next applyTo: {}) ~= nil ] whileTrue: [  ].
thisContext
  viennaReturn: (twins applyTo: {primes}) ] value ].
space ← {} asOrderedCollection.
primes ← {} asOrderedCollection.
Eratosthenes ← ViennaCompositeType
  constructorName: 'Eratosthenes'
  withAll:
    {{'space'. false. (ViennaType nat1 seq)}.
    {'primes'. false. (ViennaType nat1 seq)}}.
init_Eratosthenes ← [ :s |
s  = (Eratosthenes
  applyTo:
    {({} asOrderedCollection).
     ({} asOrderedCollection)}) ].
```

図 2. 列型を使ったエラトステネスの篩の仕様から生成された Smalltalk プログラム

```
|  next primes  space twins sieve
setup twinprime10000 prime10000 |
twins ← [ :xs |
  Array new: 1024 streamContents: [:stream |
    (xs copyFrom: 1 to: xs size - 1)
      with: xs copyWithoutFirst
      do: [ :x :y |
        y - x = 2
          ifTrue: [stream nextPut: {x. y}]]]].
setup ← [ :x |
space ← (2 to: x) asArray.
primes ←OrderedCollection new].
next ← [ space
  ifEmpty: [ nil ]
  ifNotEmpty: [ | x |
    x ← space first.
    sieve value: x.
    primes add: x]].
sieve ← [ :x |
space ← space select: [ :y| y \\ x > 0]].

prime10000 ← [
setup value: 10000.
[next value notNil ] whileTrue.
primes ].
twinprime10000 ← [
setup value: 10000.
[ next value notNil ] whileTrue: [  ].
twins value: primes ].
space ← {}.
primes ← OrderedCollection new.
```

図 3. 生成されたプログラム（図 2）を人手で改善した Smalltalk プログラム

ラムによる prime1000 および twinprime1000 の実行時間を表 4 に示す．VDMJ および VDMTools によるインタプリタ実行に対しては優位な処理速度が実現されている．人手による Smalltalk プログラムと比較すると prime10000 で 17 倍以上，twinprime10000 で 18 倍以上の実行時間となっており，生成について改善の余地が残されている．

4.2. 集合操作

space および primes を列型ではなく集合型を使って，同様の仕様を記述した．VDM-SL 仕様のほとんどは列型の機能を対応する集合型の機能に変更することで記述することができたが，大きな変更をおこなった部分を図 4 に示す．twin 関数については，集合の内包表記を使ってより簡潔な記述になった．列を使った記述では自然数が昇順で並べられていたために候補列の先頭要素を取り出していたが，集合の場合には候補集合から最小値を求める必要があるため，min 関数を新たに定義した．min 関数は，引数のあらゆる値について x <= y となるような引数中の唯一の要素を返す．

ViennaTalk で自動生成した Smalltalk プログラ

```
functions
    twins : set of nat1 -> set of (nat1 * nat1)
    twins(xs) ==
        {mk_(x, y) |
         x, y in set xs &
         y - x = 2};
    min : set of nat1 -> nat1
    min(xs) ==
        iota x in set xs &
            (forall y in set xs & x <= y);
```

図 4. 集合型を使ったエラトステネスの篩と双子素数を求める VDM-SL 仕様

```
twins ← [ :xs |
[ :__set |
| _set |
_set ← Set new.
__set do: [ :x |
  __set do: [ :y |
    y - x = 2
      ifTrue: [ _set add: {x. y} ] ] ].
_set ] value: xs ].
min ← [ :xs |
xs
  detect: [ :x |
    [ :_forall |
     _forall allSatisfy: [ :y | x <= y ] ]
     value: xs ]
  ifNone: [ ViennaNoMatch signal ] ].
```

図 5. 集合型を使った twins 関数および min 関数の仕様から生成された Smalltalk プログラム

表 5. 集合型を利用したエラトステネスの篩と双子素数の計算時間 (単位は ms)

	VT	VDMJ	VDMTools
prime10000	788	109,029	N/A
twinprime10000	833	112,332	N/A

VT は ViennaTalk による生成
N/A は計算が 1 時間以内に終了しなかったことを示す

ムを図 5 に示す．図 4 の VDM-SL 仕様での twins 関数の定義では集合内包表記で 2 つの集合束縛された変数を使っているが，生成された Smalltalk プログラムは標準クラスライブラリでの通常のプログラミングで利用される機能を使って 2 重ループとして生成している．min 関数についても，detect:ifNone:や allSatisfy:といった Smalltalk プログラムで頻出する機能が使われている．自動生成された変数名は可読性に難があるが，プログラムの構成自体は人手による自然な Smalltalk プログラムに近いものが生成されている．

集合型で記述された prime10000 および twinprime10000 による性能評価を図 6 に示す．自動生成された Smalltalk プログラムは VDMJ インタプリタに対して大きく上回る処理速度となっている．VDMTools インタプリタでは 1 時間経過しても処理が終了しなかったため N/A とした．人手による改善では，使用クラスや制御構造を変更した結果，列型からの改善と同様のプログラムになる．人手によって改善されたプログラムの処理時間は表 4 を参照のこと．

4.3. 写像操作

space および primes を写像型を使って，同様の仕様を記述した．大きな変更をおこなった部分を図 6 に示す．写像を使った仕様では，space を素数候補となる非 0 自然数から候補として残っているかどうかへの写像として表現した．例えば，{2 |-> false, 3 |-> true, 4 |-> false} の場合，元の候補であった {2, 3, 4} のうち 3 が候補として残っていることを表す．primes は自然数 n から n 番目の素数 primes(n) への写像である．

ViennaTalk が生成した Smalltalk プログラムを図 7 に示す．実行時間を計測したが，2 から 10000 までの間にある素数を求まったのは 生成された Smalltalk プログラムのみであった．2 から 1000 までの間にある素数の計算に要した時間を計測した．

5. 議論

Smalltalk 上の VDM-SL 環境 ViennaTalk に実装された VDM-SL 仕様から Smalltalk プログラムの自動生成器について，双子素数を求めるアルゴリズム仕様によって既存のインタプリタ実装との性能を比較した．

```
functions
  twins : map nat1 to nat1
              -> set of (nat1 * nat1)
  twins(xs) ==
    {mk_(x, y) |
     x, y in set rng xs &
     y - x = 2};
  min : map nat1 to bool -> nat1
  min(xs) ==
    iota x in set dom xs &
      (xs(x) and forall y in set dom xs &
         xs(y) => x <= y);
```

図 6. 写像型を使ったエラトステネスの篩と双子素数を求める VDM-SL 仕様

```
twins ← [ :xs |
([ :__set |
 | _set |
_set ←Set new.
__set do: [ :x |
  __set do: [ :y |
    ((y - x) = 2)
      ifTrue: [_set add: {x . y} ]]].
_set] value: xs values asSet)].
min ← [ :xs |
(xs keys asSet
  detect: [ :x |
    ((xs applyTo: {x}) and:
      [([ :_forall |
          _forall allSatisfy: [ :y |
            ((xs applyTo: {y}) not or: [(x <= y)])]]
       value: xs keys asSet)])]
  ifNone: [ViennaNoMatch signal])].
```

図 7. 写像型を使った twins 関数および min 関数の仕様から生成された Smalltalk プログラム

表 6. 写像型を利用したエラトステネスの篩と双子素数の計算時間 (単位は ms)

	VT	VDMJ	VDMTools
prime10000	19,609	N/A	N/A
twinprime10000	19,903	N/A	N/A
prime1000	287	34,766	N/A
twinprime1000	312	35,043	N/A

VT は ViennaTalk による生成

N/A は計算が 1 時間以内に終了しなかったことを示す

列型を用いた仕様，集合型を用いた仕様，写像型を用いた仕様のいずれでも ViennaTalk は 既存のインタプリタ実装よりも実行速度の面で優位であった．

VDM-SL による実行可能仕様は実行すること自体が目的ではない．インタプリタは仕様記述の妥当性の確認および動作への理解を深めるためのツールであり，プログラムの自動生成は十分に妥当性および正当性が検査された仕様記述から実装を得るためのツールである．自動生成されたプログラムがインタプリタ実行よりも処理速度が優れていることは，必ずしもインタプリタよりも優れた処理系であるということではない．本稿での性能評価では，自動生成器が生成したプログラムを最終的なプログラム実装を得るための 1 つのステップとして位置付け，処理速度面においてインタプリタ実行よりも優れた性能が得られていることを示した．

表 6 に示した写像型を利用したエラトステネスの篩と双子素数では，prime10000 および twinprime10000 は自動生成器によるプログラムのみが計算を終了した．ViennaTalk の Smalltalk プログラム自動生成器は，VDM-SL 仕様のうち実用的に実行可能な領域を広げることで，ソフトウェア開発における VDM の有用性を高める一助となりうると考えられる．

自動生成されたプログラムの表現については，少なくとも Smalltalk プログラムとして読むに耐えるソースコードが得られた．しかし人が記述したプログラムとはまだ明確な差があり，改善の余地があると考えられる．操作および関数の事前条件事後条件，および，再帰関数のメジャー関数など，現在の実装では生成されたプログラムに反映されない部分があり，自動生成器の適用領域を広げるために，これらをプログラム自動生成に反映させることも今後の課題と考えられる．

6. まとめ

ViennaTalk が基礎としている Smalltalk 環境は柔軟性が高く，メタ記述能力の高い開発環境であり，ViennaTalk はその柔軟性とメタ記述能力を使って，VDM-SL のツールをより効率的に開発するためのメタ開発環境である．本稿で示した Smalltalk プログラム自動生成は，ViennaTalk 環境の中で，VDM-SL 仕様と Smalltalk 環境を繋ぐ橋渡しとしての役割を担っている．VDM-SL インタプリタとの連携に加えて コード生成が可能になることでより自由度の高いツール開発が

可能になることを期待し，今後も ViennaTalk の開発を継続する．

7. 謝辞

本研究の一部は JSPS 科研費 (24220001) および JSPS 科研費 (26330099) の助成を受けた．

参考文献

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, ``Formal methods: Practice and experience,'' *ACM Computing Surveys*, vol. 41, no. 4, pp. 1--36, October 2009.

[2] N. Ubayashi, S. Nakajima, and M. Hirayama, ``Context-dependent product line engineering with lightweight formal approaches,'' *Sci. Comput. Program.*, vol. 78, no. 12, pp. 2331--2346, Dec. 2013.

[3] J. Fitzgerald and P. G. Larsen, *Modelling Systems -- Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998, iSBN 0-521-62348-0.

[4] K. Lausdahl, P. G. Larsen, and N. Battle, ``A Deterministic Interpreter Simulating A Distributed real time system using VDM,'' in *Proceedings of the 13th international conference on Formal methods and software engineering*, ser. Lecture Notes in Computer Science, vol. 6991. Berlin, Heidelberg: Springer-Verlag, October 2011, pp. 179--194.

[5] T. Kurita and Y. Nakatsugawa, ``The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip,'' *Intl. Journal of Software and Informatics*, vol. 3, no. 2-3, pp. 343--355, October 2009.

[6] T. Oda, Y. Yamomoto, K. Nakakoji, K. Araki, and P. G. Larsen, ``VDM Animation for a Wider Range of Stakeholders,'' in *Proceedings of the 13th Overture Workshop*, June 2015, pp. 18--32, gRACE-TR-2015-06.

[7] 小田朋宏, 荒木啓二郎, ``VDM-SL 実行可能仕様による Web API プロトタイピング環境,'' in **ソフトウェアシンポジウム** *2015* **論文集**, Jun 2015.

[8] T. Oda and K. Araki, ``Overview of VDMPad: An Interactive Tool for Formal Specification with VDM,'' in *International Conference on Advanced Software Engineering and Information Systems (ICASEIS) 2013*, Nov 2013.

[9] T. Oda, K. Araki, and P. G. Larsen, ``VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification,'' in *FormaliSE 2015*. In connection with ICSE 2015, May 2015.

[10] J. Fitzgerald, P. G. Larsen, and S. Sahara, ``VDMTools: Advances in Support for Formal Modeling in VDM,'' *ACM Sigplan Notices*, vol. 43, no. 2, pp. 3--11, February 2008.

[11] N. Battle, ``VDMJ User Guide,'' Fujitsu Services Ltd., UK, Tech. Rep., 2009.