

2003年度
卒業論文

SMART Action Language

神戸大学工学部情報知能工学科
清水 将仁

指導教官 林 晋

2004年2月17日

SMART Action language

清水将仁

要旨

UML の普及により、モデルを作成して、システムの構造を表現するのが一般的なものになってきている。UML には static と dynamic の 2 つの aspect があり、このうちの dynamic な aspect を UML モデルで表現するための Semantics が Action Semantics である。これを用いればプラットフォームに依存しないでオブジェクトの振る舞いを表現することができるようになると考えられる。これを用いる UML の実行では、オブジェクトが行おうとする処理はツール側が用意した言語 (これを Action 言語と呼ぶ) をコンパイラを用いて Action Semantics で定義されている要素を用いて構成されたモデルに変換し、このモデルを用いてシミュレーションを実行する、という順番で実行される。しかし、Action Semantics では Execution Engine と Action 言語が未定義になっており、ツールごとに自由に作成できるようになっている。

林研究室では現在、UML に準拠した組み込みシステム開発支援ツール SMART(現在 ver0.2) の開発を行っている。昨年、この研究の一環として Action Semantics 1.5 に準拠した並列 Action Execution Engine が開発された。しかし、これに対応する action 言語の設計とコンパイラの実装はまだなされていない。本研究ではこの並列 Executor に対応した Action 言語 SAL(SMART Action Language) の設計とコンパイラの実装を行った。

Action Semantics では、オブジェクトの振る舞いはデータの流によって表現されており、処理は action と呼ばれる単位で表現されている。SAL のプログラミングとは、小さな action を組み合わせて大きな action を作り上げる事であり、SAL にはそのための仕組みが多数用意されている。

目次

第1章 序論	4
第2章 Action Semantics	6
2.1 action specification	6
2.2 具体的な action の解説	7
2.2.1 Group action	7
2.2.2 Loop action	7
2.2.3 Map action	8
2.2.4 Conditional action	8
2.2.5 LiteralValue action	9
2.2.6 ApplyFunction action	9
2.2.7 CallProcedure action	9
第3章 SAL(SMART Action Language) の設計	10
3.1 SAL の目的	10
3.2 SAL の文法要素	10
3.2.1 pin handle	11
3.2.2 pin vector expression	11
3.2.3 pin expression	12
3.2.4 named action	12
3.2.5 action constructor	13
3.2.6 Literal Value action constructor	25
3.2.7 Apply Function action constructor	26
3.3 SAL のコンパイル	26
3.3.1 InputpinTable	27
3.3.2 OutputpinTable	27

3.3.3	pin handle の scope	28
3.3.4	ConnectionTable	29
3.3.5	action の包括関係	29
3.3.6	具体的な例	30
3.4	named action のコンパイル	33
3.4.1	functionTable	34
3.4.2	具体的な例	34
第 4 章	結論および今後の展開	38
4.1	結論および今後の展開	38
4.1.1	Action Semantics の残りの action の実装	38
4.1.2	プログラミング言語としての発展	38
	謝辞	39
	参考文献	40
	付録 A SAL のプログラム例	41
	付録 B FFT のアクション図	43
	付録 C SAL 文法	44

第1章 序論

UMLを用いたモデルを中心とした開発を支援するために、林研究室では SMART という開発支援ツールの研究を行っている。昨年、この研究の一環として UML Action Semantics 1.5 に準拠した並列 Execution Engine が開発された。しかし、これはアセンブラのようなものであり、この Execution Engine をターゲットにした高級言語の開発が必要であった。

そこで本研究ではこの Execution Engine をターゲットにした高級言語 SAL (SMART Action Language) の設計とコンパイラの実装を行った。

Action Semantics では、オブジェクトの振る舞いはデータの流れによって表現されており、処理は `action` と呼ばれる単位で表現されている。SAL は Action Semantics の能力を最大限に活かすことを目標として設計されたので、必然的にデータフロー言語となった。

`action` には任意の数の `InputPin`、`OutputPin` と呼ばれるピンが存在する。`action` はそのピンを通してデータの入出力を行っている。データのやり取りは `OutputPin` と `InputPin` を `DataFlow` と呼ばれるもので接続することによって実現される。`action` は `InputPin` は引数、`OutputPin` は返り値と考えると関数と言える。ただし `OutputPin`、つまり返り値を複数もつ点から、この関数は多値関数である。SAL のプログラミングとは小さい `action` を `DataFlow` で接続し、1つの大きな `action` をつくることである。

SAL ではいくつかの `action` を1つの `action` にまとめて名前をつけることが出来る。この `action` は一塊の処理を表す。この処理を `action` につけた名前を `key` に呼び出す事ができる。つまり一般のプログラム言語の関数の機能を使えるという事である。この関数は再帰呼び出しも可能となっている。

SAL は将来、主に次の2つの記述を行うための言語として SMART に組み込まれる予定となっている。

- 状態マシンの推移につく、`effect (activity)` の記述
- SMART の TDD of Models の テストの記述

ただし、テストの記述については、SAL を拡張した extended SAL が使われる事になる。

SAL の使い方としては、UML の動的側面を具体的に表すビヘイビアの一種であるオペレーションの記述子とも考えられるが、現在の SMART は、状態マシンによる記述が中心となっているため、直接オペレーションを SAL で記述することは行っていない。

第2章 Action Semantics

UMLには static と dynamic の2つの aspect があり、このうちの dynamic な aspect を UML モデルで表現するための Semantics が Action Semantics である。

Action Semantics は文法を定めるのではなく、UML モデル上で dynamic な aspect を記述するうえで満たさなければならない要件、つまり Semantics を定義している。そのため、Action Semantics に準拠するシステムは Semantics さえ満していれば、文法に関してはツール独自のものであってかまわない。

この章では、SAL のターゲットとなる、この Action Semantics について、コンパイルの説明に必要となる事項を説明する。

2.1 action specification

action Semantics では、オブジェクトの振る舞いはデータの流れによって表現されており、処理は action と呼ばれる単位で表現されている。データを action に流し込むと、その action は入力されたデータを用いて処理を行い、その結果が新しいデータとして他の action へ出力される。これは UNIX や DOS のパイプに似た処理モデルであり、データに依存しない action については逐次処理で行われても並列処理で行われてもかまわないものとされている。

一つ一つのアクションには任意の数の InputPin、OutputPin と呼ばれる pin が存在する。アクションはその pin を通してデータの入出力を行っている。データのやり取りは OutputPin と InputPin を DataFlow と呼ばれるもので接続することによって実現される。1つの OutputPin を複数の InputPin と接続する“fan out”は許されているが1つの InputPin を複数の OutputPin と接続する“fan in”は禁止されている。各 pin には型情報があり、DataFlow はその型情報が一致した pin 同士しか接続することはできない。アクションの実行順序は、データの依存関係により決定される。しかし、明示的に処理順序を決定することも可能で、Action Semantics ではそのために ControlFlow と呼ばれるものを提供している。

2.2 具体的な action の解説

以下に、Action Semantics に定義されている action の内で本研究に関わりの深い Group action、Map action、Loop action、Conditional action、LiteralValue action、ApplyFunction action の6つに加え、SAL のために新しく定義した CallProcedure action について説明を行う。

2.2.1 Group action

他の action を subaction として持つことが出来る action を composite action という。

Group action は composite action の一種で他の action をグループ化させ一つの action にまとめるための action で、C等のプログラミング言語におけるブロックが表す物に相当する。Group action それ自体は pin を持たず、内部に含んでいる action の pin に group action の外部から直接 DataFlow を繋ぐことが可能である。

2.2.2 Loop action

Loop action も Composite action の一種で反復処理を行う action である。内部には Clause と呼ばれる領域を一つもつ。Clause は内部に test と body と呼ばれる subaction を持つ。Clause はプログラミング言語における if...then... の部分に相当し、test が if... の部分に、body が then... の部分にあたる。Loop action はまず test を行い、その結果が真であれば body を実行する。body が終わったあと、再び test を行う。この繰り返しを行い、test が偽になったら繰り返しを終える。

Loop action は loop variables という OutputPin の List を持ち、test と body の action はこの OutputPin を参照できる。この Loop variables の値は1イテレーションの変数の値と考えることができる。body の subaction の Output Pin は Loop variables と同じ数、同じタイプ、同じ多重度でなければいけない。こうすることによって body の subaction の OutputPin の value が次のイテレーションの Loop variables の value となる。

Loop action はまた InputPin の List と OutputPin の List を持つがこれら

も Loop variables と同じ構成でなければならない。最初のイテレーションの時、InputPin の値が Loop variables のなり、最後のイテレーションが終わった時、その時の Loop variables の値が OutputPin の値となり Loop action の実行が終了する。

Loop action の外部の OutputPin を test や body の subaction の InputPin と接続することは出来るが、test や body の subaction の OutputPin を Loop action の 外部の InputPin と接続することはできない。これは “fan out” は許されるが “fan in” は許されないからである。

2.2.3 Map action

入力として Collection 型のデータを受け取り、Collection の要素一つ一つを subaction に適用していく action を Collection action という。

Map action は Collection action の一種で入力された Collection を要素一つ一つ (これを slice と呼ぶ) にわけ、それぞれを別々に subaction に渡して実行し、その出力を再び Collection にまとめて出力する action である。2つ以上の Collection を入力として受け取った場合は、その Collection はサイズが同じ Collection でなければならない、subaction へは同じ場所の各 Collection の要素が一つにまとめられ、subaction へ渡される。

Map action の出力は subaction の出力を集めた Collection となり結果として入力と同じ数、同じサイズの Collection となる。

2.2.4 Conditional action

Conditional action は Composite action の一種で条件分岐を行う action である。Loop action と同様に内部に Clause を持つのだが、Loop action とは違い複数の Clause を持つことが出来る。Conditional action は内部の Clause の test を concurrent に実行し、その test の結果のうち最初に真を返した Clause の body を実行し、その結果を Conditional action の出力とする。よって全ての Clause の body は同じ型、同じ多重度の pin を持つことになる。いずれかの Clause の body が実行されると、他の Clause の body は実行されない。また test を実行する順番を指定することも出来る。

Conditional action 自体は InputPin をもたず、内部の Clause には Conditional

action の外部の action から直接 DataFlow を繋ぐことになる。

2.2.5 LiteralValue action

LiteralValue action はシステムの primitive なデータを出力する action である。しかし、どのデータをシステムの primitive なデータにするかは設計者にまかされている。

2.2.6 ApplyFunction action

ApplyFunction action は基本的な算術計算を行うときに使用する action である。しかし、どの算術計算をシステムに取り入れるかはシステムの設計者にまかされている。

2.2.7 CallProcedure action

Action Semantics には定義されていない SAL 用に作った action。action から Procedure を呼ぶことで、プログラミング言語における関数の役割を果たす。

Procedure とは UML モデル上の実行の単位で action の集合である。action と同様に任意の数の InputPin と OutputPin を持ち、与えられた value に処理をし、出力する。

第3章 SAL(SMART Action Language)の設計

これより SAL についての説明を行う。まず SAL の目的を、次に SAL の文法要素を、最後に SAL のコンパイルについて説明する。

3.1 SALの目的

SAL は、UML Action Semantics の action を従来のプログラミング言語程度の複雑性で記述するために設計された言語である。

SMART では主に次の2つの記述を SAL で行うことになる。

1. 状態マシンの推移につく、effect (activity) の記述。
2. SMART の TDD of Models の testcase の記述。ただし、これには extended SAL という SAL を拡張した言語が使われる。

3.2 SALの文法要素

SAL の記述の文法要素のうち、基本となるのは、次のものである

1. pin handle: pin につけた仮の名前。
2. pin vector expression: action は多数の output pin をもつが、その各 pin の値を並べた vector を表す式。
3. pin expression: outputpin の値を表す式。
4. named action: 名前のついた Procedure。
5. action constructor: LoopAction, GroupAction など Action Semantics に定義されている action を組み立てるための文法要素。

以下、この5つの要素についての説明を行う。

3.2.1 pin handle

action は複数の InputPin と OutputPin を持つことは 2.1 においてすでに述べたが、SAL においてはプログラム中で pin を参照するのに、コンパイル時にのみ使われる一時的な名前を使う。これを pin handle と呼ぶ。

たとえば、FFT の butterfly 演算を入力 2 つ、出力 2 つの named action として定義するとすれば、次のようになる

```
Function Butterfly(Smi:Complex,Smntwoi:Complex,Vmi:Complex)
output output1:Complex, output2:Complex
{
    output1 <- Smi + Smntwoi;
    output2 <- (Smi - Smntwoi) * Vmi;
}
```

このプログラムをコンパイルすると、Butterfly という入力を 3 つ、出力を 2 つ持つ action が作られるのだが、このプログラムでは、入力の 1 つ目を Smi、2 つ目を Smntwoi、3 つ目を Vmi という pin handle で参照する事になり、出力の 1 つ目を output1、2 つ目を output2 という pin handle で参照する事になる。また入力の pin の type、出力の pin の type すべてが共に Complex である事も示している。

3.2.2 pin vector expression

action は、複数の InputPin、OutputPin を持つことから、多値関数 (multivalued function) であるといえる。つまり返り値が 2 以上ありえる。

たとえば、SAL で Butterfly(2,3,-i) (i は虚数単位) と書くと、Butterfly action の三つの input pin に、Smi: 2, Smntwoi:3, Vmi: $-i$ という値を流し入れたときの output1, output2 に現れる値の vector を表す。つまり、[output1:5, output2: i] という vector である。

この Butterfly(2,3,-i) が pin vector expression(略して pve) である。

3.2.3 pin expression

pve が output pin の vector の値の表現であったように、pin expression (pe) は、OutputPin 一個の値の表現である。

たとえば、vector [pin1:v1, pin2:v2,...] の pin handle pin2 で参照されるピンにおける値を参照するには、-> という記号を使い [pin1:v1, pin2:v2,...]->pin2 と書く。値は、v2 となる。

さきほどの、Butterfly(2,3,-i) の例では次のようになる。

- Butterfly(2,3,-i)->output1 = 5
- Butterfly(2,3,-i)->output2 = i

Butterfly(2,3,-i) は、長さ 2 の vector なので、これを次のように数字の参照に変えてもよい。

- Butterfly(2,3,-i)->0 = 5
- Butterfly(2,3,-i)->1 = i

pe があるべき場所に “->x” が省略されて書かれる (見た目は pve のようになる) とコンパイラは自動的に “->0” を補う。これによって例えば + のような OutputPin が一つしかない action の pe を書く時、少し簡単になるし、プログラムの見た目も分かりやすくなる。

先ほどの Butterfly(1+1,3,i) は、足し算を行う action を f とすると、

- Butterfly(f(1,1),3,i)

であり、これを pe の書き方を正確に反映して書くと、

- Butterfly(f(1->0,1->0)->0,3->0,i->0)

となる。

3.2.4 named action

SAL のプログラミングとは小さい action から、大きい action を作り上げることである。SAL では、その一つの方法として、一般のプログラム言語の関数にあたる named action というという要素を用意した。

named action はユーザが定義して使える action であり、名前がついている。この action は、複数の subaction を持ち、入力に対して内部の subaction を利用して処理を行い結果を出力する。つまり複数の action を使って行う処理を一つの action にまとめることが出来て、その処理を action の名前を参照することで呼び出すことが出来るようになる、関数の働きをする action である。

named action は関数のように実際に使用する場所とは違う場所で定義することになる。また再帰呼び出しも可能となっている。

named action の定義は次のようになる

```
Function actionname(argument1:type,argument2:type ...)
output result1:type,result2:type, ...
{
    ...
};
```

named action の宣言部の持つ意味は 3.2.1 で pin handle を説明する際にすでに述べたのでここでは省略し、ここでは本体部に関しての説明を行う。本体部には named action の処理の内容を記述するのだが、その処理はデータを内部に持つ action にある順序で流し、加工する事で実現される。つまり本体部にはどのような action があり、その action 同士をどのように繋ぐかという事が記述される。Action Semantics では複数の action をまとめてグループ化し、一つの action にまとめる action として Group action が用意されている事は 2.2.1 ですすでに述べたが named action の本体部でする事とはまさに Group action をつくる事であり、その内部の action の DataFlow を決定することである。

{...}の部分は Group action constructor という文法要素であり、先ほど述べた処理はこのなかで記述される。Group action constructor の詳細は後に説明する。

3.2.5 action constructor

SAL のプログラミングとは小さい action を組み合わせ、ある処理を実現する大きい action を作り上げる事であるという事はすでに述べたが、核となる小さな action は Action Semantics によって定められている。それらの action を構築するための文法要素が action constructor である。

SAL は将来的には、Action Semantics の action すべてに対応する action constructor を持つことになるが、ここでは現在実装済みである Group Action con-

structor、Map action constructor、Loop action constructor、Conditional action constructor、LiteralValue action constructor、ApplyFunction action constructor の5 つについてこれから説明を行う。各 constructor の説明は次のような構成となっている。

概要 action constructor の用途を簡単に述べる

pin action constructor が表す action がどのような pin を持っているかを述べる。

書き方 action constructor の文法の簡単化した説明。

詳細 action constructor のそれぞれの特徴について詳しく書く。

Group action constructor

概要 Group action constructor は、Group action を構築する action constructor である。Group action constructor の内部では、今記述している Group action がどのような action で構成されているか、また、今記述している Group action の内部の action の pin はどの pin とつながっているのか、という事を記述していく。

つまり Group action constructor では action の塊にデータを流した時の処理が描かれることになり様々な action の本体はこの Group action constructor となる。

pin Group action は pin を持たない。

書き方 次の2種類の記述方法がある。

1. 通常ブロック {...}
2. 出力指定ブロック sequential {...}

詳細 FFT(付録 A 参照)には次の Group action constructor が出てくる。これを使って説明を行う。

(fft1) FFT の定義本体

{

```

sequential:
PinHandle _n;
let [_n] be size(argument);
loop
(...
  V = roots_of_unity(_n),
...)
if (greater(log2(_n),m))
....
};
result <- bitreverse(@preaction -> 1);
};

```

(fft2) loop 文の本体

```

{
PinHandle _leftS, _rightS;
@post_m <- @pre_m+1;
@post_S <- 略
@post_V <- Thin(@pre_V,@pre_k);
@post_k <- @pre_k * 2;
};

```

(fft3) @post_S の値の指定

```

sequential {
let [_leftS, _rightS] be Cut(@pre_S);
map (left:Complex <- _leftS, right:Complex <- _rightS,
     k:Complex <- @pre_V)
output Seven:Complex, Sodd:Complex
{
...
};
Shuffle;
};

```

ブロックの中に書けるものは次の文法要素である。

1. sequential 修飾子
2. pin handle 宣言
3. pin handle 設定
4. binding statement
5. pve
6. pe
7. action constructor
8. named action の名前 (sequential ブロックのみ)

このうち sequential 修飾子と pin handle 宣言は必ずブロックの先頭 (ただし sequential 修飾子のほうが先) に書くことになっている。

まず、sequential 修飾子について説明を行う。

sequential 修飾子 Group action の内部の action は concurrent に実行されるので、Group action constructor も concurrency を意識した書き方となっている。しかし、アルゴリズムには sequential なものも多く、この場合 concurrency を意識した書き方をすると冗長になる事が多い。Group action constructor ではブロックの先頭で "sequential:" と宣言することで、C 言語風の直列逐次的な書き方ができるようになる。

具体的には次の 2 つの書き方ができるようになる。

1. 直前の statement が pve か action constructor named action 名であれば、そこから出来る action の OutputPin を "@preaction -> x" として参照できる。
2. 直前の statement が pve か action constructor か named action 名であり、そこから出来る action の OutputPin の数が named action の名前によって参照される action の InputPin の数と等しかった場合、

```
{sequential:
  ...
  pve か action constructor か named action 名;------(1)
  named action 名;------(2)
  ...
```

```
};
```

という記述ができる。これは (1) から出来る action の OutputPin をそのまま (2) から出来る action の InputPin に繋ぐという状況を意味している。

p.14の(fft1)とp.15の(fft3)でこの書き方を使用している。(fft1)では"result <- bitreverse(@preaction -> 1);"の@preaction -> 1で直前のLoop action constructor で作成される Loop action の OutputPin の2つ目(番号は0が最初であるから)を参照している。sequential ブロックでなければ Loop action constructor と "result <- bitreverse(@preaction -> 1);" は並列に実行されるべきであるのでこのような書き方は許されていない。

(fft3)では Map action constructor で作成される Map action の OutputPin 2つを Shuffle(InputPin 2つ、OutputPin 1つの action) の InputPin にそのまま繋げている。

次に pin handle 宣言と pin handle 設定について説明を行う。

pin handle 宣言と pin handle 設定 pin handle 宣言は次のように行う。

```
Output name1:type,name2:type,...;
```

pin handle 設定は次のように行う。

```
let [pinhandle1,pinhandle2,...] be pve;
```

pin handle の scope は、そのブロック内である。だから、次のように同じレベル、あるいは、子や孫のレベルでも参照できる。

```
{
  Output _x:int;
  <- goo(_x);
  loop ....
  {
    <- foo(_x);
  };
}
```

pin handle の scope 内で例えば次のように pin handle 設定を行うとする。

```
let [_x,_y] be foo(...);
```

すると foo の output pin の 1 つ目を `_x` で、2 つ目を `_y` で参照できるようになる。

p.14 の (fft1) では、4 行目の pin handle 設定により、Loop action constructor 中の 2 ヶ所の `_n` から、size の OutputPin の 1 つ目を参照することになる。これにより、一つの OutputPin が 2 ヶ所の InputPin に接続されるという “fan out” が実現される。

また、p.15 の (fft3) を見てみると、Cut の二つの OutputPin は、図 (付録 B を参照) をみると “fan out” してないが、`map (...)` の (...) の 2 箇所で、二つの pin をつなく必要があるために、pin handle を設定している。もし、これを

```
sequential {  
  map (left:Complex <- Cut(@pre_S)->1, right:Complex <- Cut(@pre_S)->2,  
      k:Complex <- @pre_V) ...
```

と書くと、Cut の action を二つ作ることになってしまい、意図した物とは別の action が組み立てられてしまうが、pin handle により `Cut(@pre_s)` に名前をつけることでこれを防げるのである。

コンパイラは Input pin に名前をつけるので Input pin handle というべきものがコンパイル時には生成される。しかし上記の 2 つの使い方を見ればわかるように、Input pin handle をプログラム中で宣言して使うという機会はない。このため、pin handle 宣言を “Output pin handle 名” という風に Output pin handle を意識させる書き方にした。

次に Binding statement について説明を行う。

Binding statement p.14 の (fft1) の `result <- bitreverse(@preaction -> 1);` を Binding statement という。これは「Input pin handle `result` が指す InputPin に `bitreverse` の OutputPin を DataFlow で繋げ」という指示を表している。しかし、`result` をたとえば、出口の値を表す変数とみなせば、これを

```
result := bitreverse(@preaction -> 1);
```

と読むことができ、Pascal や C 言語の代入文になり、実際、それに近い感覚でプログラミングを行うことができる。

次に `pve` がプログラム中でかかれた時の pin の接続について説明する。

pve に関する pin 接続 例えば `Butterfly(2,3,i)` は pve になるが、その `2, 3, i` は、いわゆる式、expression である。これらは値、`2, 3, i` を表すと考えればよいが、実際は `LiteralValue` action constructor でこれらの値を出力する action がつくられる。つまり `Butterfly(2,3,i)` は「`2, 3, i` という三つの action の OutputPin を `Butterfly` の三つの InputPin に、その順番につないでできた action の OutputPin の値の vector」となる。これにより、「`2, 3, i` という三つの action の OutputPin を `Butterfly` の三つの InputPin に、その順番につなぐ」という pin のつなぎ方が指示されていることになる。

最後に出力指定ブロックについて話をする。

出力指定ブロック `Group action constructor` は通常ブロック `{...}` と出力指定ブロック `sequential {...}` の 2 種類があると説明したが、実は `{...}` の部分が `Group action constructor` で、出力指定ブロックとは、`{...}` の最後の pve の OutputPin をそのまま自身の OutputPin とする pve である。

p.15 の `(fft2)` と `(fft3)` でこの機能が使われている。`(fft3)` のブロックの最後は `Shuffle;` で終わっており、この `Shuffle` の OutputPin を `(fft2)` の `@post_S` で参照する InputPin に繋ぐことになる。

Map action constructor

概要 `Map action constructor` は `Collection` の一つ一つの要素に対して、ある一定の処理を施したい時に使用する `Map action` を記述する為の action constructor である。

pin `Map action` は次の 4 種類の pin を持つ。

1. action の外部と接続する pin
 - `argument(InputPin)`
 - `result(OutputPin)`
2. action の内部のみで使う pin
 - `subinput(OutputPin)`
 - `suboutput(InputPin)`

書き方

```
map (argument1:type <- pe,argument2:type <- pe, ...)
output result1:type -> Output pin handle, ...
{
  略
};
```

詳細 FFT(付録 A 参照) には次の Map action constructor がでてくる。これを使って説明を行う。

```
map (left:Complex <- _leftS, right:Complex <- _rightS, k:Complex <- @pre_V)
output Seven:Complex, Sodd:Complex
{
  @slice_Seven <- @slice_left + @slice_right;
  @slice_Sodd <- (@slice_left - @slice_right) * @slice_k;
};
```

left, right, k が argument の pin handle 名の指定であり、Seven, Sodd が result の pin handle 名の指定である。Map action は入力される要素も出力される要素も Collection であるので、pin の type には Collection の pin handle 名の一つの要素の type を指定する。この pin handle 名の指定のもとに、subinput, suboutput が derive され、全体として、次のような pin handle が作られる。

- subinput: @slice_left, @slice_right, @slice_k (derived)
- suboutput: @slice_Seven, slice_Sodd (derived)
- argument: left, right, k (given)
- result: Seven, Sodd (given)

derive された @slice_left は、left という argument の一つの slice を表す。result の slice も同様である。

Collection の要素に対して施したい処理をこの subinput, suboutput を使って Group action constructor の中で記述する事となる。

```
map (left = _leftS, right = _rightS, k = @pre_V) ...
```

の `_leftS`, `_rightS`, `@pre_V` はこの Map action の argument の `left`, `right`, `k` に繋ぐべき、OutputPin を表す pin handle である。

Map action constructor において宣言される pin handle は、そのブロック内でのみ有効であるべきである。よって map action の result pin を同じブロックか、もしくは外側のブロックにある action の InputPin に繋ぎたい場合、Output pin handle に map action の result pin を関連づけなければならない。このために `output result:type -> Output pin handle` の `-> Output pin handle` 部 (以後これを result binder と呼ぶ) は用意されている。

しかし、FFT の Map action constructor は sequential ブロック (Group action constructor で説明) の中で使われているので result binder を省略した書き方となっている。

Loop action constructor

概要 Loop action constructor は繰り返し処理をしたい時に使う Loop action を記述する為の action constructor である。

pin LoopAction は次の 5 種類の pin を持つ。

1. action の外部と接続する pin
 - loopVariableInput (InputPin)
 - result(OutputPin)
2. action の内部のみで使う pin
 - loopVariable (OutputPin)
 - bodyOutput (OutputPin)
 - testOutput (OutputPin)

書き方

```
loop(name1:type <- pe,name2 <- pe, ...)  
output @result_name1 -> Output pin handle, ...  
if(...)
```

```
{
  略
};
```

詳細 FFT(付録 A 参照) には次の Loop action constructor がでてくる。これを使って説明を行う。

```
loop
(m <- 0,
 S <- argument,
 V <- roots_of_unity(_n),
 k <- 1)
if (greater(log2(_n),m))
{
  略
};
```

Loop action constructor は、 m , S , V , k を元に、次のように pin handle を作る。

- loopVariable: @pre_m, @pre_S, @pre_V, @pre_k (derived)
- loopVariableInput: m , S , V , k (given)
- result: @result_m, @result_S, @result_V, @result_k (derived)
- bodyOutput: @post_m, @post_S, @post_V, @post_k (derived)

testOutput の pin handle はユーザが testOutputPin を参照できないので作られない。

```
if (greater(log2(_n),m))
{
  略
};
```

の if の部分は、clause の条件、

```
{
  略
};
```

の部分が clause の本体である。本体の役割は、loop variable に次の値を流すことだから、

```
{
  PinHandle _leftS, _rightS;
  @post_m <- @pre_m + 1;
  @post_S <- 略
  @post_V <- Thin(@pre_V, @pre_k);
  @post_k <- @pre_k * 2;
};
```

のように、loop variable に値を流す、binding statement が 4 個置かれた Group action constructor が書かれることになる。@pre は、UML の OCL を真似た記法で、m という loop variable の 1 iteration の前の m の値、@post は、その iteration が終わった時の値という気持ちを表している。

この Loop action constructor が、コンパイルされると、できるのは、Loop action ではないことに注意しないとイケない。この Loop action constructor には、

```
(m <- 0,
 S <- argument,
 V <- roots_of_unity(_n),
 k <- 1)
```

という部分があり、0, roots_of_unity(_n), 1からはそれぞれに対応した action が生成される。argument は pin handle なので action は作られない。if ... 以後の部分を parse して作った Loop action と 3 つの action が Loop action constructor では作られることになる。

FFT の Loop action constructor も Map action constructor と同様に sequential ブロックで使われているので Loop action の result binder も省略した書き方となっている。

Conditional action constructor

概要 Conditional action constructor は条件分岐を行いたい時に使う Conditional action を記述する為の action constructor である。

pin Conditional action は次の pin を持つ。

1. action の外部と接続する pin
 - output(OutputPin)
2. action の内部のみで使う pin
 - bodyOutput (OutputPin)
 - testOutput (OutputPin)

書き方

```
conditional
output result1:type -> Output pin handle, ...
{
  [sequential:]
  if(...) <- 条件部と呼ぶ
  then{略}; <- 処理部と呼ぶ
  if(...)
  then{略};
  ... 略
};
```

詳細 これは FFT(付録 A 参照) には現れないので、次の例を使って説明を行う。

```
conditional
output result:int -> a
{
  sequential:
  if(x > 1)
  then{
    result <- Fib(x - 2) + Fib(x - 1);
  };
  if(x > 0)
  then{
    result <- 1;
  };
};
```

```

};
if(true)
then{
    result <- 0;
};
};

```

条件部が Clause の test を、処理部が Clause の body を表すのは Loop action と同様である。result から BodyOutput の pin handle “result” が作られる。処理部では必ず作られた BodyOutput の pin handle に対して Binding statement を書かれなければならない。

Conditional action の body に sequential: と宣言する事で、Clause の test が行われる順番を上から下に行く事を指定することが出来る。この例では $(x > 1) \rightarrow (x > 0) \rightarrow (\text{true})$ の順番で test が行われる事になる。この指定がなければ test は concurrent に実行される事になる。

3.2.6 Literal Value action constructor

概要 Literal Value action constructor はシステムの primitive なデータを出力する Literal Value action を記述するための action constructor である。

pin Literal Value action は次の pin を持つ。

1. action の外部と接続する pin
 - result(OutputPin)

書き方

- number(int)
- number.number(real)
- true(bool)
- false(bool)

詳細 SAL では primitive なデータとして “int”、“real”、“bool” を用意した。

3.2.7 Apply Function action constructor

Apply Function action constructor はシステムの組み込みの算術計算を提供する Apply Function Action を記述するための action constructor である。

pin Apply Function action は次の pin を持つ。

1. action の外部と接続する pin
 - argument(InputPin)
 - result(OutputPin)

書き方

1. 2項演算子
 - $x + y$ 、plus(x,y)
2. 単項演算子
 - $- y$ 、log(x)

詳細 “ $x + x$ ” は演算であるが、SALではこれを plus(x,x) という関数が記述された、ととらえ入力2つ、出力1つの action を作る。

3.3 SALのコンパイル

この節では、前節で説明した SAL 文法を、どのようにコンパイラが処理するかを説明する。

SAL をコンパイルするには action の pin をどのようにつなぐかという点が処理の大部分を占める。

action の pin をつなぐためには プログラム中に現れる InputPin のハンドルを管理するテーブル、OutputPin のハンドルを管理するテーブル、OutputPin をどの InputPin につなぐかを記載したテーブルが必要である。

この3つのテーブルを InputpinTable、OutputpinTable、ConnectionTable と名づける。

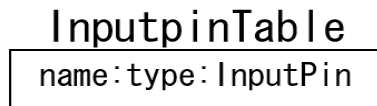


Fig 3.1: InputpinTable

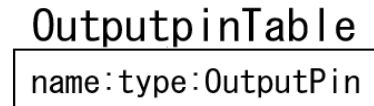


Fig 3.2: OutputpinTable

3.3.1 InputpinTable

InputpinTable は InputPin のハンドルの名前、タイプ、どの InputPin を指しているか、という情報のタプルを要素として持ち、Binding Statement(a <- b; 等) が現れた時、右辺を解析した結果判明する OutputPin を どの InputPin に繋ぐかという情報を、Binding Statement の左辺の識別子をキーに参照し、InputPin を得るために使う。InputpinTable に要素を登録するタイミングは、次の2つである。

1. named action の宣言時
2. 一部の action constructor (現在の時点では loop、map、conditinal) の宣言時

Fig 3.1 がこのテーブルの図である。

3.3.2 OutputpinTable

OutputpinTable は OutputPin のハンドルの名前、タイプ、どの OutputPin を指しているか、という情報のタプルを要素として持ち、pin expression 中で識別子が現れた時、その識別子をキーに OutputpinTable を参照し、OutputPin を得る為に使う。OutputpinTable に要素を登録するタイミングは次の3つである。

1. named action の宣言時
2. 一部の action constructor (現在の時点では loop、map) の宣言時
3. Pin Handle 宣言時

Fig 3.2 がこのテーブルの図である。

3.3.3 pin handle の scope

pin handle はプログラム中で変数のように参照されるので、scope の問題を解決しなくてはならない。InputpinTable、OutputpinTable 共に次のルールを採用することで scope を実現した。scope 毎の作られるテーブルをサブテーブルと呼ぶ事にする。

1. ブロックの始まりで今までのサブテーブルの集合に新しいサブテーブルをぶら下げる
2. ブロック内で宣言された新しい pin handle 最後尾のサブテーブルの要素として登録する
3. ブロックの終わりでは最後尾のサブテーブルを削除する
4. Table から要素を取得する時は最後尾のサブテーブルから探し始め、そのサブテーブルで見つからなければ 1つ前のサブテーブル、まだ見つからなければさらに一つ前...と検索し、最初に見つかった要素を返す

この方法で scope が実現できるという事を次の例を使って解説する。

```
{
  Output x:int;-----(1)
  {
    Output x:int;-----(2)
    {
      Output x:int;-----(3)
      c <- x;
    };-----(4)
    b <- x;
  };-----(5)
  a <- x;
};
```

(1) 2つ目のブロックが始まる前、(2) 3つ目のブロックが始まる前、(3) 3つ目のブロックに pin handle 'x' が登録された時、(4) 3つ目のブロックが終わった後、(5) 2つ目のブロックが終わった後の OutputpinTable は Fig 3.3 のようになる。

このプログラムの場合、c <- x; の x は 6 行目の x、b <- x; の x は 4 行目の x、a <- x; の x は 2 行目の x を参照したい。これは x を参照する時に OutputpinTable

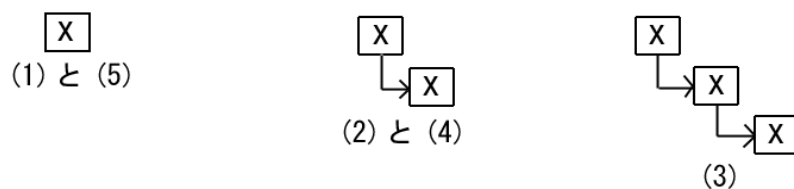


Fig 3.3: scope

の最後のテーブルから検索を始めて、最初に見つかった `x` を当てることで実現できる。

3.3.4 ConnectionTable

Connection Table は OutputPin と InputPin のペアを要素として持ち、コンパイルの最後でそれぞれを DataFlow で繋ぐ為に使用する。繋ぐべき OutputPin と InputPin が判明した時点 (Binding Statement や pve 等) でそのペアを要素としてこのテーブルに追加する。Connection Table はプログラム中では要素を追加するのみで参照する事がないので InputpinTable や OutputpinTable と異なり scope は必要無い。よって ConnectionTable は 1つのテーブルで実現できる。Fig 3.4 がこのテーブルの図である。

以上で pin の繋ぎ方の説明を終える。

3.3.5 action の包括関係

SAL をコンパイルする処理のうち pin の繋ぎ方以外で大きなウェイトを占めるのは action の包括関係の把握である。Map action や Loop action、Group action 等は他の action を内部にもつ事が出来るので、どの action を持っているのかという情報を知っていなければならない。これには actionsTable というもので対応した。

actionsTable は要素として action を持つ。actionsTable もブロックの始まりでは Outputpin Table と同様に新しいテーブルをぶら下げるが、しかしこれは “{...}” 内で宣言された action をブロックの終わりで一つの Group action にまとめるのにこの構造が便利であるからであり scope の為ではない。Fig 3.5 がこのテーブルの図である。

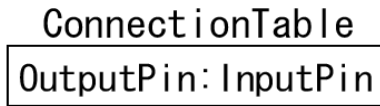


Fig 3.4: ConnectionTable

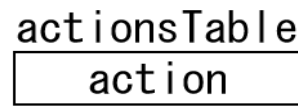


Fig 3.5: actionsTable

3.3.6 具体的な例

この節では、テーブルの使われ方を、実例を使って説明する。

```
//Tableの使い方の説明の例
Function foo(a:int,b:int) output c:int
{
  Output x:int;
  let [x] be a + 5;
  {
    Output y:int;
    let [y] be x * b;
    c <- y;
  };
};
```

Fig 3.6 は foo の action 図、Fig 3.7 はそれぞれのテーブルの図である。

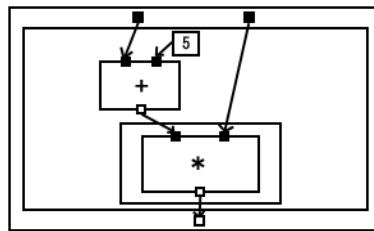


Fig 3.6: action figure of foo

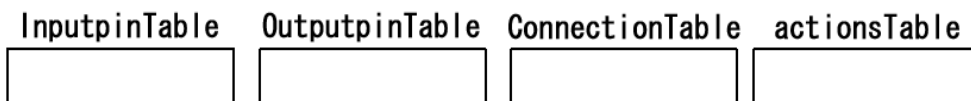


Fig 3.7: state of 0

一行目からは、次の3つがわかる。

InputpinTable
x: int:foo. SuboutputPin[0]

OutputpinTable
a: int:foo. SubinputPin[0]
b: int:foo. SubinputPin[1]

Fig 3.8: state of 1

OutputpinTable
a: int:foo. Argument [0]
b: int:foo. Argument [1]
x: int:

Fig 3.9: state of 3

1. a は type が int で Function Action の SubInputPin の一個目を指している
2. b は type が int で Function Action の SubInputPin の 2 個目を指している
3. c は type が int で Function Action の SubOutputPin の一個目を指している

よって a と b を OutputpinTable に、c を InputpinTable に登録する。Fig 3.8 はこの時のテーブルの状態の図である。

3 行目からは、次の事がわかる。

1. x という名前の outputpin のハンドルの type が int である

よって x を OutputPin に登録する。この時点では x がどの OutputPin を指しているかという情報は確定していないので登録されない。Fig 3.9 はこの時のテーブルの状態の図である。

4 行目からは、次の 5 つの作業をしなければいけないことがわかる。

1. a + 5 から plus という ApplyFunctionAction を作り、actionsTable に plus を登録する
2. 5 から LiteralValueAction[5] を作り、actionsTable に 5 を登録する
3. a を OutputpinTable から引くと FunctionAction の SubInputPin の一個目だという情報が得られるので、FunctionAction の SubInputPin の一個目を plus の InputPin の一個目に繋ぐという情報を ConnectionTable に登録する
4. LiteralValueAction[5] の outputPin を plus の InputPin の 2 個目に繋ぐという情報を ConnectionTable に登録する
5. x が plus の OutputPin を指しているという情報を追加する。

Fig 3.10 はこの時のテーブルの状態の図である。

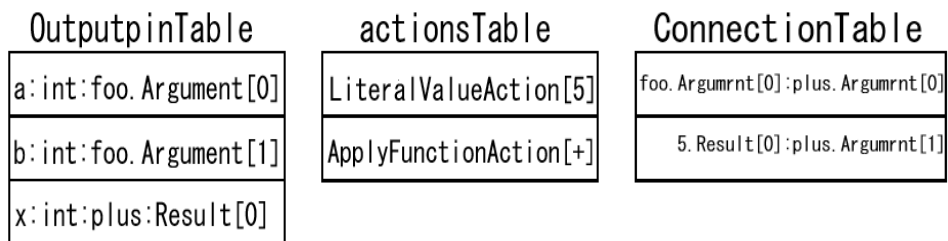


Fig 3.10: state of 4

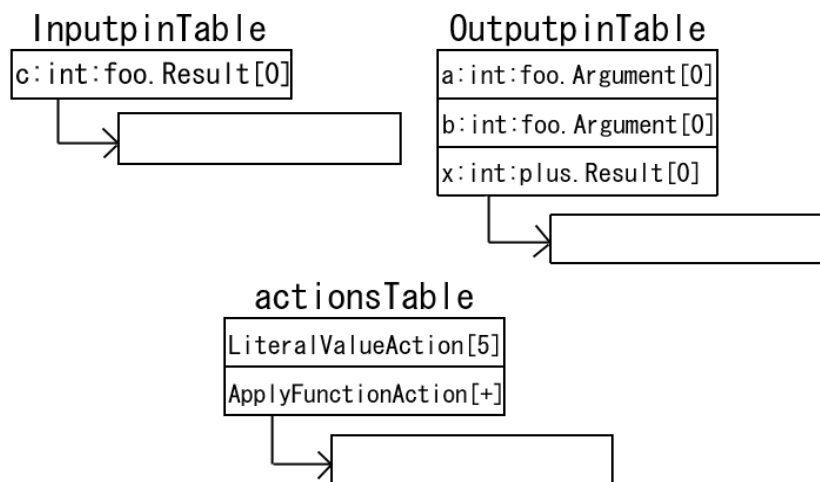


Fig 3.11: state of 5

5 行目では、新たなブロックが始まったので InputpinTable、OutputpinTable、actionsTable に新しいサブテーブルをぶら下げる作業をする。Fig 3.11 はこの時のテーブルの状態の図である。

8 行目の Binding Statement `c <- y;` から次の 3 つの作業をする事になる。

1. Binding Statement の左辺は InputPin になるので InputpinTable から `c` を参照し、それが Function Action の Sub Output の 1 つ目を指しているという情報を取得する
2. 右辺は OutputPin になるので OutputpinTable から `y` を参照し、それが times の OutputPin を指しているという情報を取得する。
3. times の OutputPin と Function Action の Sub Output の 1 つ目のペアを ConnectionTable に登録する Fig 3.12 はこの時のテーブルの状態の図である。

ConnectionTable
foo. Argument[0] : plus. Argument[0]
5. Result[0] : plus. Argument[1]
Plus. Result[0] : times. Argument[0]
foo. Argument[1] : times. Argument[1]
times. Result[0] : foo. result[0]

Fig 3.12: state of 8

9行目では、ブロックの終わりなので InputpinTable、OutputpinTable、actionsTable から最後尾のテーブルを削除するのだが、この際、actionsTable では削除するテーブルに登録されている action を group action にまとめ、1つ前のテーブルに登録するという作業を行う。Fig 3.13はこの時のテーブルの状態の図である。

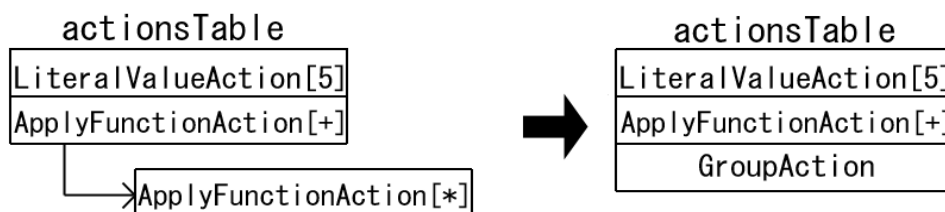


Fig 3.13state of 9

10行目で foo という名前をキーに Function Action [foo] を Map に登録し、ConnectionTable から要素を一つずつ取り出し、それぞれを DataFlow で接続して foo のコンパイルが終了する。

3.4 named action のコンパイル

named action は一般のプログラム言語の関数の働きをするのだが、特殊な処理が必要となるので、その仕組みをこれから説明する。

named action ブロックはコンパイルが終了すると、Procedure になる。UML モデルでは action は様々なレベルにグループ化されるが、その一番高いレベルのグループが Procedure である。UML モデルではこの Procedure を method の body として呼び出す事になる。何故 named action ブロックをコンパイルした結

果 Procedure を作るのかについては後述する。

この Procedure を管理するテーブルとしてfunctionTableを作る。

3.4.1 functionTable

functionTable は Procedure とそれにつけた名前の情報のタプルを要素として持ち、プログラム中で、すでに登録された識別子を名前として持つ named action が現れた時、そこで作られる Call Function Action に持たせる Procedure を、識別子を key に参照するする為に使われる。

再帰呼び出しを可能にするために、named action の宣言時で Procedure をつくり、functionTable にその Procedure を登録する。この段階では InputPin と OutputPin は決定しているが、その中身はこれから定義していく事になるので空のままである。

named action のブロックの中で再帰呼び出しが行われた場合、Call Function Action をつくり、それに Procedure を持たせるのだが、named action の宣言時に Procedure はすでに作られているので、結果として再帰呼び出しも可能となる。

プログラムのコンパイルは上から下へ sequential に行われるので、named action を定義する前に named action を参照したい場合、プロトタイプ宣言を行う。

プロトタイプ宣言:

```
prototype actionname(type of pin1,...) output type of pin1,...;
```

Call Function Action で使う Procedure はその時点で中身まで決まっている必要がないので先に Procedure の枠が決まっていればよい。よってプロトタイプ宣言で先に型枠を作ることで、named action を定義する前に named action を参照できるようになる。

3.4.2 具体的な例

以上の操作を例をもって説明する。

```
Prototype hoo(int) output int;  
Function foo(a:int) output b:int  
{  
... (略)
```

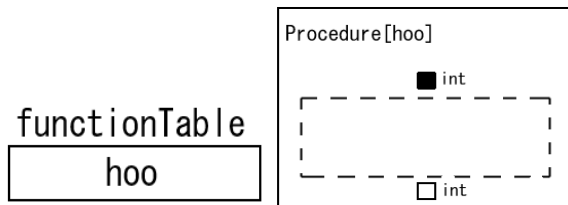


Fig 3.14: state of 1

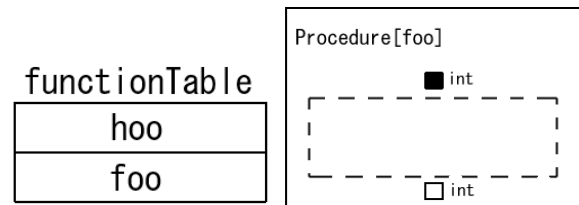


Fig 3.15: state of 2

```

...hoo(x);
...hoo(y);
...foo(z);
...(略)
};
Function hoo(a:int) output b:int
{
...(略)
};

```

1 行目のプロトタイプ宣言から次の作業を行う。

1. hoo と名づけられたタイプが整数型の InputPin と OutputPin を一つずつ持つ Procedure を作り、functionTable に登録する

hoo の中身は後で定義されるのでこの段階では空である。Fig 3.14 はこの時のテーブルの状態と Procedure[hoo] の図である。

2 行目の named action 宣言から次の作業を行う。

1. foo という名前の Procedure が登録されているかどうか functionTable を調べる
2. 登録されていなければ、foo と名づけられたタイプが整数型の InputPin と OutputPin を一つずつ持つ Procedure を作り、functionTable に登録する。

foo の中身はすぐ後のブロックで作られるのでこの段階では空である。Fig 3.15 はこの時のテーブルの状態と Procedure[foo] の図である。

3 行目、4 行目はとばして 5 行目、hoo(x) から次の作業を行う。

1. Call Function Action[hoo_1] を作る

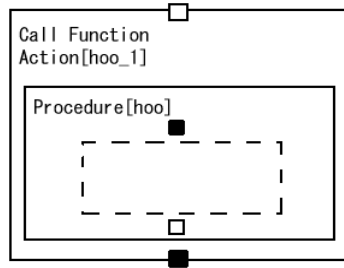


Fig 3.16: state of 5

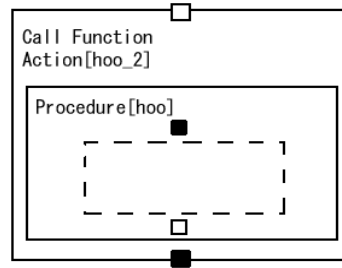


Fig 3.17: state of 6

2. functionTable から hoo という名前を持つ Procedure を参照し、Call Function Actin[hoo_1] にそのポインタを持たせる
3. actionsTable に Call Function Action[hoo_1] を登録する

Fig 3.16 はこの時の Call Function Action[hoo_1] の図である。
6 行目、hoo(y) から次の作業を行う。

1. Call Function Action[hoo_2] を作る
2. functionTable から hoo という名前を持つ Procedure を参照し、Call Function Actin[hoo_2] にそのポインタを持たせる
3. actionsTable に Call Function Action[hoo_2] を登録する

Fig 3.17 はこの時の Call Function Action[hoo_2] の図である。
7 行目、foo(z) から次の作業を行う。

1. Call Function Action[foo_1] を作る
2. functionTable から foo という名前を持つ Procedure を参照し、Call Function Actin[foo_1] にそのポインタを持たせる
3. actionsTable に Call Function Action[foo_1] を登録する

Fig 3.18 はこの時の Call Function Action[foo_1] の図である。
8 行目から次の作業を行う

1. foo の中身の action の定義が終わったので Procedure にその action を登録する

Fig 3.19 はこの時の Procedure[foo] の図である。
9 行目から後では次の作業を行う。

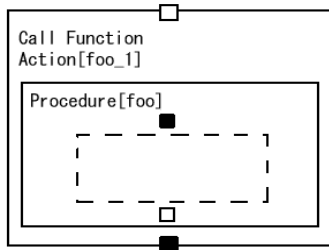


Fig 3.18: state of 7

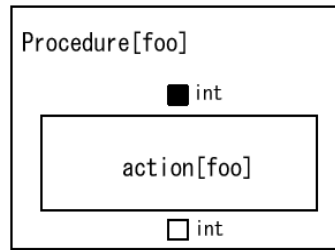


Fig 3.19: state of 8

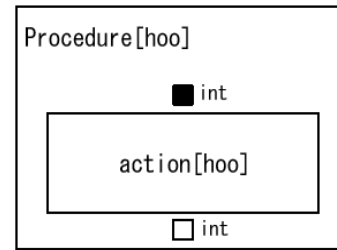


Fig 3.20: state of 9

1. hoo という名前の Procedure が登録されているかどうか functionTable を調べる
2. すでに登録されているので、この後のブロックを解析し、その結果出来た action を hoo に登録する

Fig 3.20 はこの時の Procedure[hoo] の図である。

当初、Call Function Action は Group Action に InputPin、OutputPin をつけた action として考えていた。こうした場合、同じ action を参照する Call Function Action を 2 つ以上作るとその内部の action の InputPin に対して fan in が起こってしまう。Action Semantics では fan in は禁止されているので当然 Action Semantics に準拠した Executor は動かなかった。

そこで、Call Function Action を作る際、その内部の action とその action に関する DataFlow をコピーして fan in を解決するという方法を考えたのだが、こうした場合、再帰的定義を行うと、自らの action の中に、自らを展開することになり無限ループに陥ってしまい、コンパイルが出来なくなってしまうという問題がでてきた。

これらの問題を、Call Function Action を Procedure を持ち、実行時にはその Procedure の実行をシステムの外部に委託し、受け取った結果をそのまま出力する action として実装することで解決した。Call Function Action は内部の Procedure の実行をシステムの外部に委託する時、InputPin の値を引数として渡し、その結果を返回值として受け取る。よって Procedure とは直接 DataFlow で繋がれないので “fan in” が起こることはない。また Call Function Action には Procedure のポインタを持たすので再帰的定義を行っても、無限ループに陥ることはない。この方法を採用することで action を関数のように使う事が出来るようになった。

以上で named action のコンパイルに関する説明を終わる。

第4章 結論および今後の展開

4.1 結論および今後の展開

本研究の結果、SMART に並列 Executor を組み込む際に必要となる Action Language の文法とコンパイラの実装の基礎ができた。また、これは多値関数でシステムを記述するデータフロー言語になっている。

SAL は Action Semantics に準拠した Executor に対応する高級言語だが、Action semantics の全ての要素を実装するにはいたっておらず、まずはこれに対応しなければならない。また、SAL は多値関数でシステムを記述するデータフロー言語という新しい側面も持っており、この可能性を探る事も同時に行わなければならない。よってまずは次の2つが今後のやるべき事になる。

4.1.1 Action Semantics の残りの action の実装

Attribute への書き込みやイベントの発生等、Action Semantics で定められている残りの action に対する実装を行う。

4.1.2 プログラミング言語としての発展

SAL は返り値を複数もつ関数を使ってシステムを記述するデータフロー言語であるが、この言語でシステムをどのように、またどこまで記述できるのか、その結果はまだ見えていない。この可能性を探るためには、例えば SAL に class 概念を導入し、オブジェクト指向プログラミングができる言語としての可能性を追求する必要がある。

謝辞

筆者がこの研究を進めるにあたり、理論的基礎から実装に至るまでの全課程において熱心な御指導、御教授をいただきました神戸大学工学部情報知能工学科 林晋教授に心から感謝いたします。

また、SAL のコンパイラの実装において、多くの御指摘、アイデアをいただき、Action Executor への Call Function Action の実装を行っていただきました神戸大学大学院自然科学研究科 花山健二氏に感謝いたします。

最後に平素より様々な場面でお世話になりました神戸大学大学院自然科学研究科 潘沂冰氏、薛世宗氏、森健司氏および神戸大学工学部情報知能工学科 佐藤真沙美氏に感謝いたします。

参考文献

- [1] Object Management Group, “Action Semantics for the UML”, http://www.kc.com/as_site/home.html (2001)
- [2] Andrew W.Appel, “modern compiler implementation in Java”, *Cambridge University Press* (2002)
- [3] Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman 著/原田賢一 訳, “コンパイラ 原理・技法・ツール” サイエンス社 (1990)
- [4] Alfred V.Aho, Jeffrey D.Ullman 著/土居範久 訳, “コンパイラ”, 倍風館 (1986)
- [5] 中田育男, “コンパイラ”, オーム社, (1995)
- [6] 結城浩, “Java 言語で学ぶデザインパターン入門”, ソフトバンクパブリッシング株式会社 (2001)
- [7] 花山健二, “UML Action Semantics 実行系の開発”, 卒業論文, 神戸大学 (2003)

付録A SALのプログラム例

SALの文法の説明をする前に具体的な1例としてFFT(Fast Fourier Transform)のプログラムを紹介する。FFTとはDFT(Discrete Fourier Transform):時間領域, 周波数領域ともに離散化されたFourier変換を高速に行う計算法であるが、このうちのバタフライ演算とよばれる計算は、パラレルに行うことができ、並列動作可能なAction Semantics Executorの例として非常に有用であるので、本論文ではこの例を中心にして説明を行っている。

```

Function FFT(argument:Vector) output result:Vector
{
  sequential:
  Output _n:int;
  let [_n] be size(argument);
  loop
  (m:int <- 0,
   S:Vector <- argument,
   V:Vector <- roots_of_unity(_n),
   k:int <- 1)
  if (greater(log2(_n),@pre_m))
  {
    Output _leftS:Vector, _rightS:Vector;
    @post_m <- @pre_m + 1;
    @post_S <- sequential {
      let [_leftS, _rightS] be Cut(@pre_S);
      map (left:Complex <- _leftS, right:Complex <- _rightS,
          k:Complex <- @pre_V)
      output Seven:Complex, Sodd:Complex
      {
        @slice_Seven <- @slice_left + @slice_right;
        @slice_Sodd <- (@slice_left - @slice_right)*@slice_k;
      };
      Shuffle;
    };
    @post_V <- Thin(@pre_V,@pre_k);
    @post_k <- @pre_k * 2;
  };
  result <- bitreverse(@preaction -> 1);
};
以下略

```

Fig. A.1:sample program of SAL(FFT)

付録B FFTのアクション図

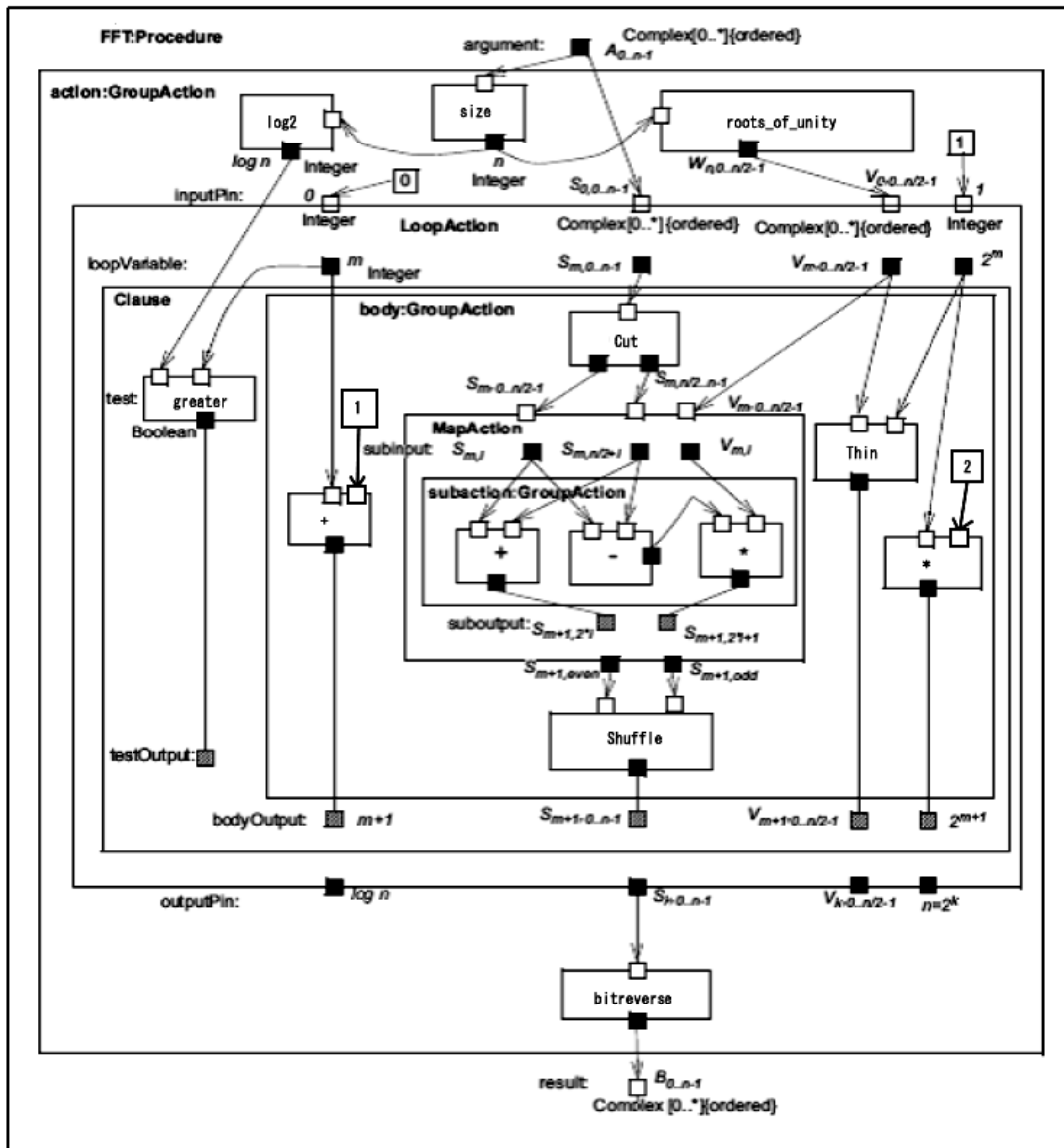


Fig. B.1: action figure of FFT

付録C SAL文法

```
goal ::= preblock;
```

```
preblock ::= preblock1 block;
```

```
preblock1 ::= PROTOTYPE ID LPAREN (VOID | ID (COMMA ID)*) RPAREN  
            OUTPUT (VOID | ID (COMMA ID)*) SEMI preblock1  
            | //empty  
            ;
```

```
block ::= actionconstruct SEMI block  
        | mainblock SEMI block  
        | //empty  
        ;
```

```
mainblock ::= MAIN groupact;
```

```
actionconstruct ::= FUNCTION ID LPAREN (VOID | ID COLON ID (COMMA ID  
COLON ID)*) RPAREN OUTPUT (VOID | ID COLON ID (COMMA ID COLON ID)*)  
groupact;
```

```
groupact ::= [SEQUENTIAL] LBRAZE [SEQUENTIAL COLON] [OUTPUTPIN ID COLON  
ID (COMMA OUTPUTPIN ID COLON ID)*] statements RBRAZE;
```

```
statements ::= (statement SEMI)*;
```

```
statement ::= binding_statement  
           | pinhandleset_statement
```

```

        | aspve
        | ID
        | ID LPAREN pve1 RPAREN
    ;

pinhandleset_statement ::= LET LSQBR ID (COMMA ID)* RSQBR BE pve;

pve ::=  aspve
        | expr
    ;

binding_statement ::= [ATMARK] ID LARROW pe;

pe ::=  pve [RARROW (NUMBER | ID)];

expr ::=  expr (PLUS | MINUS | TIMES | DIVIDE | MOD | GREATER | LESS
| BITSHIFTRIGHT | BITSHIFTLEFT) expr
        | MINUS expr
        | LPAREN expr RPAREN
        | (NUMBER | DOUBLENUM | TRUE | FALSE | [ATMARK] ID)
        | ATMARK PREACTION RARROW NUMBER
        | ID LPAREN pe (COMMA pe)* RPAREN
    ;

aspve ::=  loop_pve
        | map_pve
        | groupact
        | conditional_pve
    ;

loop_pve ::= LOOP LPAREN ID COLON ID LARROW pe (COMMA ID COLON ID
LARROW pe)* RPAREN OUTPUT ATMARK ID [RARROW ID] (COMMA ATMARK ID
[RARROW ID])* IF LPAREN pe RPAREN groupact;

```

```
map_pve ::= MAP LPAREN ID COLON ID LARROW pe (COMMA ID COLON ID LARROW
pe)* RPAREN OUTPUT ID COLON ID [RARROW ID] (COMMA ID COLON ID [RARROW
ID])* groupact;
```

```
conditional_pve ::= CONDITIONAL OUTPUT ID COLON ID [RARROW ID] (COMMA
ID COLON ID [RARROW ID])* LBRAZE (IF LPAREN pe RPAREN THEN groupact
SEMI)* RBRAZE;
```