MetaEdit+ DSM のジェネレータ定義について

DEFINING GENERATORS WITH MetaEdit+

全てのコードを一字一句記述することが賢明でないことは周知のこと。いくつかの機能を実装することで、それらコードの類似性を見出して、繰り返されるパターンに気付くことができる。毎回決まりきったコードを実装したくは無いので、繰り返される個所を自動生成するといった、コード生成の自動化の考えに至ることは必然。この資料では、どのようにしてコード自動生成機能が MetaEdit+ で実現できるかについて紹介する。

1 ジェネレータの動作原理

コードジェネレータはオートマトン（automaton＝自動装置）として、３つの作業を実施する。

１）モデル内にアクセスしてナビゲート
２）モデルから情報を抽出
３）その情報を特定のシンタックスで出力

これら３つのステップについて、状態遷移図を用いて説明する。Fig 1 はステートマシンの一部で、カード支払いの読み込みについての仕様。モデルにはスタートエレメント（黒丸）と、２つの状態（四角形）、２つの遷移（矢印）がある。

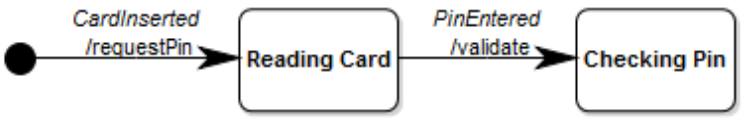

Fig 1. A partial state transition diagram

モデルへのアクセス方法

モデルへのアクセスは言語のメタモデルに基づいて行われる。ジェネレータはモデリング言語内で使用されるコンセプトのみを基にしてモデルをアクセスしてナビゲートできるということ。Fig 2 はステートマシン（Fig 1）のコンセプトを示すメタモデル。ジェネレータは Start オブジェクトなど特定のエレメント（ルートとなる）から、あるいは State オブジェクトなど特定のオブジェクトタイプを探して、あるいは様々な接続のタイプ（‘Transition’リレーションシップなど）から、ナビゲーションを開始する。

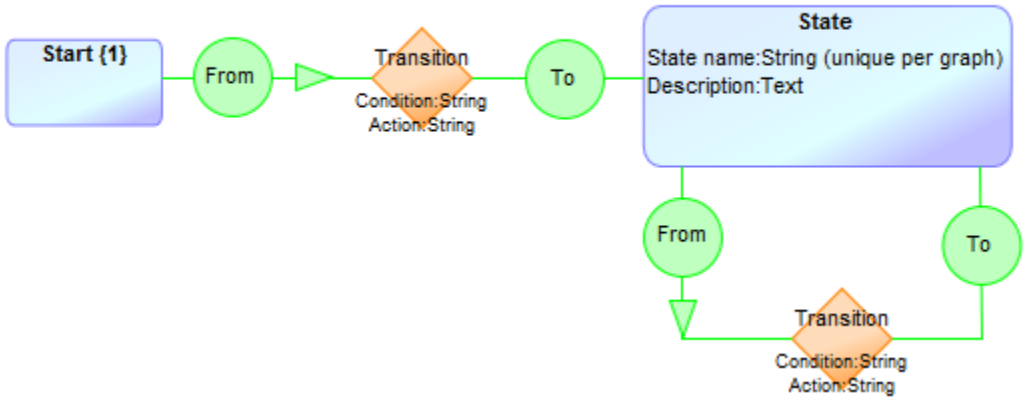

Fig 2: A metamodel of the state transition diagram used in Fig 1

Start ステートからモデルアクセスとナビゲーションを開始するジェネレータは、MetaEdit+ では以下のように定義される。

```
foreach .Start {
    do >Transition.State {
        /* generator handling the accessed state here */
    }
}
```

ここでジェネレータのスクリプトは、メタモデル内で宣言された表現（‘Start’、‘Transition’、‘State’など）を使用する。メタモデルでスタートのステートは一つだけと定めているので（Fig 2 内の‘Start (1)’ を参考：メタモデル上に 1 つ以上持てないという制約を設定している）、ジェネレータは唯一つのスタートステートを必ず見出せる。もしそのようなスタートオブジェクトがモデル図になければ、このジェネレータは何もしない。そして次に 2 行目で、トランジションリレーションシップに従ってナビゲーションし、スタートの次に来る最初のステートにアクセスする。

MetaEdit+ のジェネレータエディタは、ジェネレータの記述とデバッグをサポートする。ジェネレータの記述にはモデリング言語（メタモデル）のコンセプトを直接使用できることに加え、生成されたコードからモデルの該当箇所を直接開くことができる。これによりジェネレータの定義はアジャイルに実施できる。Fig 3 は、上述のモデル内をナビゲーションするスクリプトとジェネレータエディタの画面。

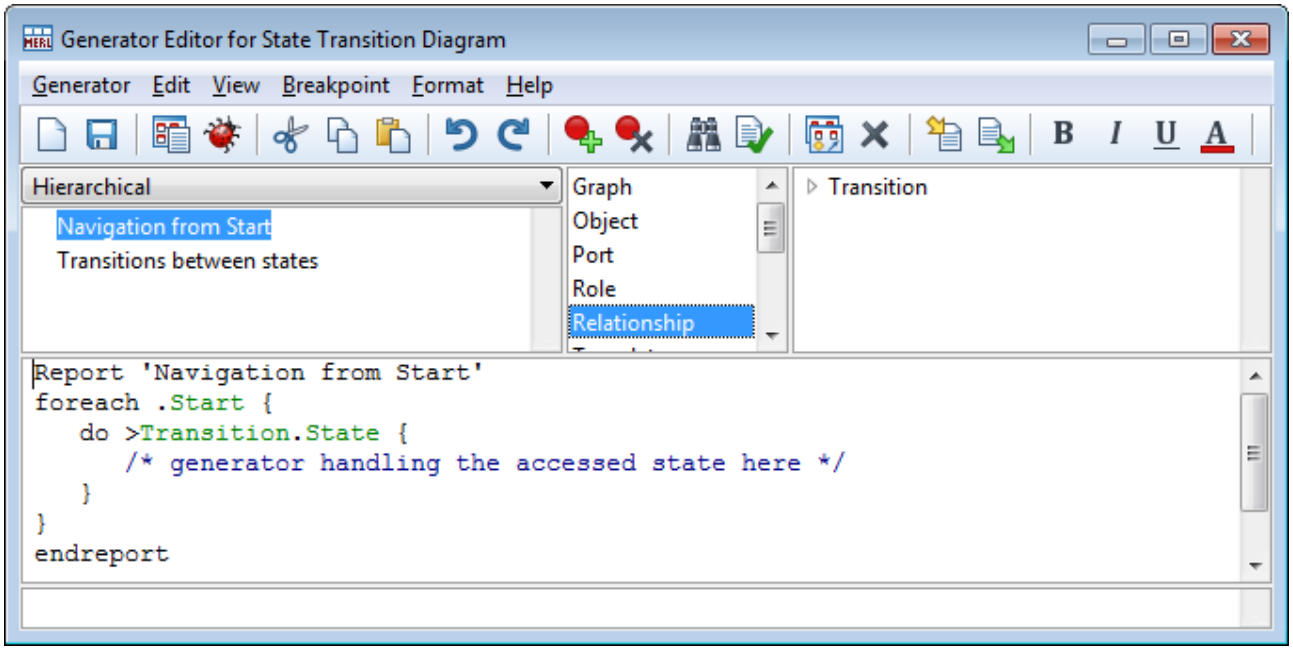

Fig 3: Generator script in MetaEdit+ Generator Editor

Start エレメントからモデルアクセスする代わりに、以下のように state から開始するようにジェネレータを定義することもできる：

```
foreach .State {}
```

トランジションをベースにするならジェネレータは以下のようになる：

```
foreach >Transition {}
```

更にモデル内の特定の値によってモデル内ナビゲーションやモデルエレメントへのアクセスをさせるなら、以下のように特定コンディションのトランジションのみをアクセスさせることもできる：

```
foreach >Transition; where :Condition; {}
```

あるいは別の（サブ）ダイアグラム内に定義されるサブステートを持つステートにアクセスするなら：

```
foreach .State {
    if decompositions; then
        /* generator for the state with a substate here */
    endif
}
```

これらのオプションによってジェネレータは接続やサブモデルを、それらの深さなどに応じてナビゲーションすることができるし、またナビゲーションやアクセスの順を決めることもできる。一般に、モデル内のアクセスとナビゲーションに用いるモデルデータの殆どは、そのモデルのデザイン情報と同じであり、以下のモデルデータも含めることができる。

- モデルエレメントの特定の位置、サイズや他のエレメントとの配置（あるエレメントが大きなエレメント内にある場合など）
- モデルの様々な管理データ、生成タイミングやバージョン、その他の情報
- ソートの順、独自の順番付けなど

異なるモデル言語で構成される複数のモデルでも、ナビゲーションは共通の原則をベースにできる（MetaEdit+ はマルチ言語対応であり、状態遷移図など単一モデルのアクセスに限定されるのではないということ）。各モデリング言語が同じメタモデルを使用するなら、ジェネレータは個々に分離されたモデルを統合して扱える。各モデルが相互に分離されているなら、ジェネレータで生成時に統合できる。例えばストリングのマッチングやモデル間のリンクを示すマーキングをモデル内に持たせるなど。

モデルデータの抽出

モデルデータの抽出もまたメタモデルに基づく。コードジェネレータはメタモデル内にフォーマルに定義されたもの以外は取り出せない。シンプルな例では、各モデルのエレメントからモデル内の値を含む一定のコードを生成する。代表的な例として、以下のようなテンプレートベースのコード生成がある：

```
foreach .State {
    'case '
    :State name
    ':' newline
}
```

2 行目でジェネレータは決まったテキスト（case）を生成する。そして 3 行目では、対象のステートオブジェクトからステート名を抽出する。4 行目では：を追加して改行。このジェネレータが Fig 1 のモデルで起動されると、ジェネレータ内のテンプレートを基に、各ステートに 1 行のコードが以下のように生成される。

```
case Reading card:
case Checking pin:
```

モデルのナビゲーションとアクセスを組み合わせるとジェネレータは以下のようになる：

```
1 foreach .State {
2   :State name;
3   :Documentation;
4   do ~From>Transition {
5       :Trigger
6   }
7 }
```

ここで１，４行目はモデルのナビゲーション方法について、２，３，５行目はモデルへのアクセス。モデルをナビゲートしてアクセスすることで、シーケンシャルな情報、関数コール、スイッチケース構造、状態遷移テーブルなどに従ったコードが生成できる。

モデルからコードを生成

モデル内をナビゲートしてアクセスされるデータはコード生成に利用される。それらの出力にジェネレータによって追加情報が加わってフレームワークのコードや、RTOS やライブラリなどターゲット環境にアクセスする（例えば Android のサービスとか）。またデータは生成される言語に応じたフォーマットに変換することもできる。例えば先に紹介したジェネレータも、以下の例では 3 行目に%var がありモデル内の名前が適切な変数名に変換される。（例えば ‘my_first_state’ とモデル内に記述されてもコード上は ‘myfirstsate’に自動補正させるとか）

```
1 foreach .State {
2   'case '
3   :State name%var
4   ':' newline
5 }
```

そしてステート名からスペースが除去され、生成される結果は以下のようになる

```
case Reading_card:
case Checking_pin:
```

要約すると、ジェネレータの動作はメタモデル、モデリング言語とそのコンセプト、セマンティクス、ルールが前提となって導かれ、ターゲット環境ごとの入力構文で決定される（アセンブラコードの生成ならそれに従ったシンタックスで）。この資料末尾の Appendix に補足情報として、４つの異なるジェネレータを紹介する。（Assembler, C, Python, XML）

2 ジェネレータの設定方法

ジェネレータを構築する前に、何を生成させるかを知る必要がある。そのため、モデルと生成されるコードの両方のサンプルを参照する必要がある。

リファレンスにするコードとモデルを用意する

リファレンスになるコードは、最も経験豊富なプログラマにお願いするのが良い。そして他の開発者に教える目的で実装されるようなコードスタイルを依頼すること。さもなければ様々な特殊なトリックが盛り込まれ、そのドメインのアプリケーションに汎用に使える標準的なコードでは無くなるので。仮に、後日に何らかの理由でジェネレータを断念する場合でも、少なくとも標準化、汎用化可能なエキスパートのコードが残ることを期待でき

る。経験豊かな開発者のコードならジェネレータの構築は加速化される。様々なコーディング作法や標準に関する議論も軽減できるので。

そしてコードが用意できれば、ドメインスペシフィックモデリング言語で、そのアプリケーションをモデル化する。一つのモデル内に３～４種の主なモデリングコンセプトの一エレメントを持つようなものを。例えば Fig 1 にあるような最小限のものから始めて、後からエレメントを追加して拡張できる。そしてモデルを作りながら必要な情報を提供できるかをチェックする。資料末尾の Appendix では、４つの異なるモデリング言語と、それらから生成されるコード（Assembler, C, Python, XML）を紹介している。

ジェネレータ構築 シンプルな手順

モデル内の情報は、オブジェクト、リレーションシップ、プロパティに展開されている。そしてアウトプットは同じセマンティクスに加えて、生成する言語の構文に関連する内容やジェネレータ自身で追加できる中身がある。ただ明らかなことは、成果物内で最も重要なバライアビリティはモデルから得られるということ。ジェネレータはモデルに対して一定に働き、同じモデルから生成される出力はいつでも同じ。モデル内に無いバライアビリティを生成されるコード内に加えることは無い。

それゆえ出力されたコードを見れば固定されたテキスト部分（必ず存在する）とモデルからの値とを識別することができる。そしてこれら対極なるものは、ジェネレータの本質的な処理としてモデルからの情報によって取捨される部分や、モデル内の構造に応じて繰り返される部分に展開される。

ジェネレータはこれら 4 つの部分で殆どカバーされる。たとえ複雑なシステムであっても。あらゆるジェネレータに対処するために、モデルから値を得る機能が必要となる（生成させる言語のシンタックスに許容される範囲で）。
例えば、大抵の言語では名前はアルファベットとアンダースコアのみで構成される必要がある。例えモデル内で名前にスペースを使うにしても。前述の %var でステート名を変換した例のように。

ジェネレータの構築のシンプルな手順は以下の通り：

1) 期待される出力コードはペーストする（ジェネレータの完全な中身として）
2) 出力されるコード内の繰り返されるセクションを減らして一度きりにする。そのようなセクションがある各モデル構造へのジェネレータのループを介して。
3) 一つ以上の形式の選択があるセクションは、それをジェネレータコードで囲んでモデル内のコンディションに応じて正しい形式を選択させる。
4) モデル内のプロパティ値に応じて出力を差し替えて、ジェネレートされるコードにプロパティの値を持たせ、必要に応じてフィルターする

実践的には、ステップ２～４が並行して実施されていく

ジェネレータの構造化

色々な意味で、ジェネレータを実装するということはプログラム実装の特別なケースであると言える。そして同じ振舞いは様々な構造で実装できる。しかしながら、より良いと思われるいくつかの方法がある。作り易く、デバッグ、メンテナンス性に優れ理解しやすいという点で。このようなコードの構造化策の多くはジェネレータの実装にも適している。またジェネレータには独自の要件があり、とりわけモデリング言語内の複雑な構造への考慮が必要。

一般にジェネレータは小さな一つのものへと構造化される。各々がサブジェネレータを呼ぶような階層化されたジェネレータが特に良い。
ジェネレータのトップレベルの“オートビルド”のみがモデリング担当者から見えていて、これが他のジェネレータをコールすることで、各モデルから生成される様々なファイルに応じたタスクに分割される。生成されるファイルはコード、コンフィギュレーション、テストデータなど。

必要かつ可能なら、ファイルジェネレータは更にモデル内のオブジェクトタイプに応じたタスクに分割される。同一モデルセットに一つ以上のプログラミング言語が必要なら、そのサポートは同時並行した構造にできる。

ジェネレータの構造に、とても良い方法の一つはモデリング言語ごとに定義して、それぞれの主なる言語コンセプトごとにサブジェネレータを持たせるようにする。例えば、‘States’を処理する一つのジェネレータを用意して、他のもので‘Transitions’などのコードを生成させる。これの利点は、モデリング言語が変更されれば、それに伴って変更が必要になるジェネレータ部分が明白になること。また生成されるファイルを基盤にしてジェネレータを構造化することもできる。あるいはファイル内の生成されるセクションを基盤にすることもできる。

コード以外にドキュメントなども生成できる

ジェネレータで自動生成できるのはコードだけではない。コンフィギュレーションデータやテストケース、シミュレーション環境、ドキュメント、自動ビルドプロセス、その他様々な成果物を同じモデルから生成させることができる。システムレベルのモデルならハードウエアマッピングやネットワークデバイスの設定、配線設計、部品表、設置ガイド、配線キャビネットのラベルなども生成できる。単一のソース（モデル）から複数の成果物を得られるので、修正があっても唯一つの箇所で済んで、残りはツールで自動化できる。

3 SUMMARY

ジェネレータの構築は、モデルのコンセプトをコードや他の成果物に、どのようにマップするかということ。シンプルな例では、各モデル要素から、特定のフィックスされたコード内に、モデル内に記述された値を持たせて生成する。
さらに他のモデル要素とのリレーションシップや、他のモデルの情報（サブモデルや別のモデリング言語のモデルなども）、更には既存のライブラリーコードの対処をさせること

もできる。ジェネレータがモデルから情報を得る方法や、それを基にコードに変換させる方法は、生成されるコードに合わせることができる。

MetaEdit+ はジェネレータの実装や、デバッグができるツールを搭載する。ジェネレータ実装用のエディタでは、モデル内の設計データを選択して、特定フォーマットのコードを生成させることができる。そして定義されたジェネレータは、複数のモデル担当者によって開発された、様々な設計モデルからのコード生成に活用される。
ジェネレータはコードやコンフィギュレーションの生成、ドキュメントやデータディクショナリ生成、モデルの一貫性チェック、メトリクスの生成、モデルリンケージの解析や、シミュレータ、コンパイラ、デバッガなどへのエクスポートなど、幅広い用途で活用されている。コマンドスクリプトによって設計情報を様々な形式に保存することや、複数ファイルへ出力したり、外部プログラムを起動することもできる。

以下のアペンディックスでは様々なジェネレータの例として、XML、アセンブラ、Python、C コードの生成を紹介する。これらは MetaEdit+の評価版のインストール内のサンプルで更なる理解を得ることができる。

APPENDIX: EXAMPLES OF GENERATORS

いくつかのコード生成事例を紹介する。全てを紹介することは出来ないが、様々な言語（XML, Python, C, Assembler）を、様々なアプローチで生成できることを例示する。各例は、アプリケーションロジック、振る舞いを含んだ動作可能なコードを生成する（静的な構造のみを生成する程度ではないということ）。

Example 1: Generating XML

XML 生成用のジェネレータ定義は、一般にシンプル。特にモデル言語が XML スキーマに上手くマップされている場合など。両方とも同じドメインを表現することができるので。ＸＭＬの制約からＸＭＬスキーマの可読性が犠牲になるときなど、モデリング言語で上手く対処できる。そのような場合、ＸＭＬで必要な重複や表現に対する少しの追加調整をジェネレータで行える。

以下、Call Processing Language (CPL)の事例。ＣＰＬはインターネット電話サービスの制御と構成ができる。システムがどのようにコールするかなど。ＣＰＬ言語の構造は、その振舞いに上手くマップされる。それゆえＣＰＬサービスの開発担当者は Fig 4 にあるようなグラフィカルなモデルを容易に理解して正当性を確認できる。Fig 4 からモデルの利点も明らか。ＣＰＬサービス開発者でなくてもモデルを解釈できること（ＸＭＬで記述されているモデルでは解読が困難であるが）

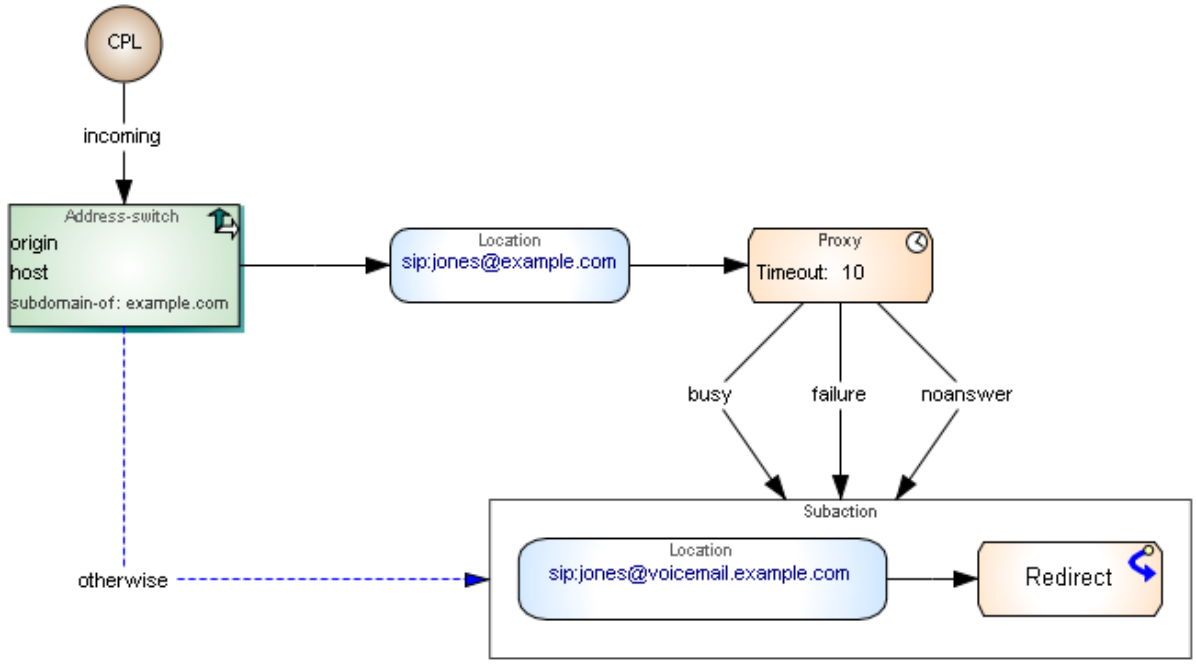

Fig 4. Redirecting Calls: A sample call redirecting service expressed in CPL.

モデリング言語はＩＰ電話サービスを規定するのに欠かせない proxy, location, signaling actions といったコンセプトを持つ。そしてジェネレーションの処理は極めて明快。なぜならＸＭＬで既に言語のコンセプトは定義されているし、モデリング時の属性値などはアトリビュートとしてＸＭＬのエレメントにあるので。ジェネレータがやることは、接続に従ってノードごとに、相当するＣＰＬドキュメント構造をＸＭＬ形式にすること。Fig 4 のモデルからジェネレータにより生成されるＸＭＬは以下の通り。

```
01  <?xml version="1.0" encoding="UTF-8"?>
02  <!-- DOCTYPE call SYSTEM "cpl.dtd" -->
02  <!-- DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL
    1.0//EN" "cpl.dtd" -->
03  <cpl>
04  <subaction id="voicemail">
05    <location url="sip:jones@voicemail.example.com">
06      <redirect />
07    </location>
08  </subaction>
09  <incoming>
10    <address-switch field="origin" subfield="host">
11      <address subdomain-of="example.com">
12        <location url="sip:jones@example.com">
13          <proxy timeout="10">
14            <busy><sub ref="voicemail" /></busy>
15            <noanswer><sub ref="voicemail" /></noanswer>
16            <failure><sub ref="voicemail" /></failure>
17          </proxy>
18        </location>
19      </address>
20      <otherwise>
```

```
21        <sub ref="voicemail" />
22      </otherwise>
23    </address-switch>
24  </incoming>
25  </cpl>
```

ジェネレータは始めにサービス仕様内の全サブアクションを探索する。この例ではひとつのサブアクション（モデル図内・右下のボイスメールの四角枠）があって、これに対して以下４～８行目のコードを生成する。

```
04  <subaction id="voicemail">
05    <location url="sip:jones@voicemail.example.com">
06      <redirect />
07    </location>
08  </subaction>
```

このボイスメールサブアクションはロケーションのエレメント（5行目）とリダイレクトのエレメント（6 行目）を宣言し、リダイレクション（かけ直し）を自動的に有効にする。

サブアクションを生成し終わったら、ジェネレータはメインのコールプロセッシングのアクションを生成する。サービスの開始（Fig 4：ＣＰＬの茶丸）から、"Incoming"のリレーションシップを介して Address-switch のオブジェクトへとモデル（仕様）内を巡回する。そして Address-switch ノードのプロパティをジェネレートされるコードのアトリビュートとして生成する（以下、10~11 行目）

```
10    <address-switch field="origin" subfield="host">
11      <address subdomain-of="example.com">
```

ジェネレータは続いてメインの流れを矢印に従って次のオブジェクトに進み、location の定義を生成する（以下 12 行目）

```
12        <location url="sip:jones@example.com">
```

さらに進んで proxy の処理が生成される（13~17 行目）。始めにタイムアウトのアトリビュート（13 行目）。そして３つのオールタネイトの接続がプロキシエレメントから生成される。

```
13          <proxy timeout="10">
14            <busy><sub ref="voicemail" /></busy>
15            <noanswer><sub ref="voicemail" /></noanswer>
16            <failure><sub ref="voicemail" /></failure>
17          </proxy>
```

最後に example.com 以外の発信元アドレスの場合用の 20~22 行目が生成される。

```
20      <otherwise>
21        <sub ref="voicemail" />
22      </otherwise>
```

生成されるコードは、設計段階で妥当性がチェックされた完全なサービス。なぜならモデリング言語にドメインのルールを組込むことで、サービスの開発者は妥当で上手く構成された設計モデルしか組めないので。また一貫性が守られていることをチェックして、不備があれば担当者に指摘する機能をモデリング言語に持たせることもできる。例えば足りない設定に対する警告表示などで（call redirect の設定忘れを指摘するとか）。このようなルールはドメイン固有なのでドメインスペシフィック言語内でのみ実現できる。

Example 2: Generating Assembler for 8-bit Microcontrollers

同様にモデル内をナビゲーションするアプローチの別の例。しかしながら、少々複雑な事例。8 ビットマイコン用のアセンブラコードの生成。このデバイスにはボイスメニュー機能があって、様々なホームオートメーション機器の遠隔操作ができる。電灯のオン・オフ、室温調整、エアコン制御など。システムは 8 ビットマイコンを介してプログラマブル（アセンブラライクな言語で）。この場合、コードサイズやメモリに関する課題を考慮に入れてジェネレートする必要がある。この辺が、上述のＸＭＬ生成事例との違い。

開発者はメモリアドレスへのアクセス、計算処理、比較とジャンプなどの基本動作に加えて、メニューアイテムを音声で読み込むなどのボイスメニューならではの操作も扱う必要がある。これらのコンセプトはドメインスペシフィックモデリング言語内に直接持つ。そのような言語のデザインモデル例。Fig 5

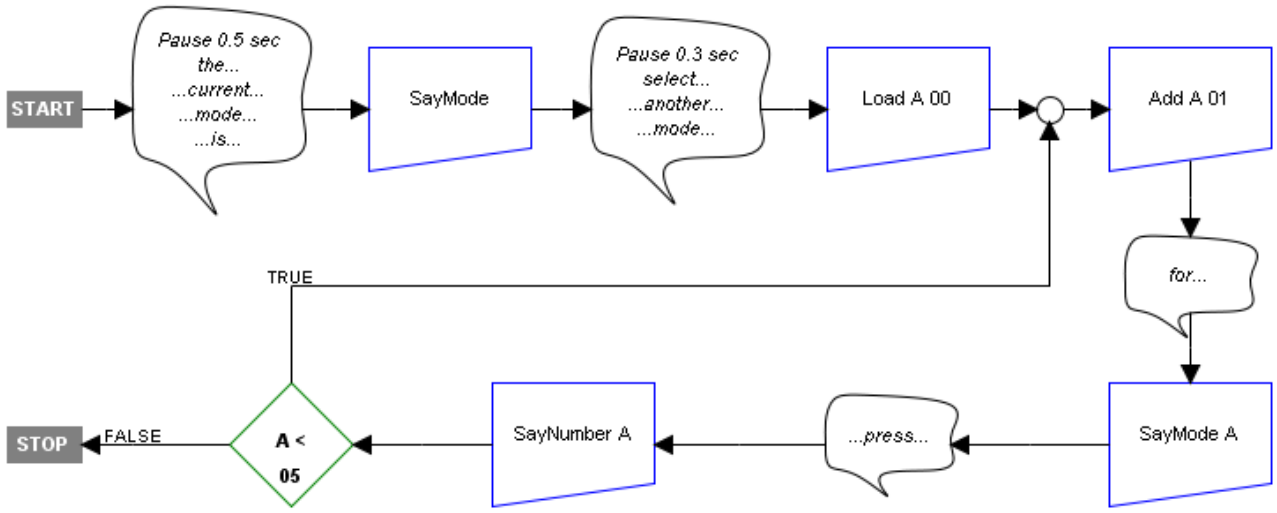

Fig 5. Setting Parameters: The figure shows a design for setting one home automation parameter in a voice menu system.　ホームオートメーション機器のボイスメニューのパラメータ設定

以下、Fig 5 のモデルから生成されるコード例。このモデルとサンプルコードでは変数名を用いているが、製品、開発組織内では実メモリアドレスにするかもしれない。

```
01   Say   0x01   5    'Pause 0.5 sec'
02   Say   0x02   1    'the...'
03   Say   0x03   2    '...current...'
04   Say   0x04   1    '...mode...'
05   Say   0x05   1    '...is...'
```

```
06  SayMode
07  Say  0x06  3   'Pause 0.3 sec'
08  Say  0x07  2   'select...'
09  Say  0x08  3   '...another...'
10  Say  0x04  1   '...mode...'
11  Load A  00
12 :10_663
13  Add A  01
14  Say  0x09  1   'for...'
15  SayMode A
16  Say  0x00  1   '...press...'
17  SayNumber A
18  Test A <  05
19  If
20  Jump 10_663
```

モデル内に記述されるアクションの実行フローがコードジェネレーションの基本となる。このモデリング言語のメタモデルは比較的シンプル。各基本の動作タイプ（話す、メモリーアクセス、比較）は個別のモデリングコンセプトとして実装されていて、これらコンセプト間のリレーションシップとして、実行フローや条件分岐がある。
各モデリングコンセプトはコマンドタグ、パラメータ、コンディションなどの設計上のアトリビュートを扱う。このアセンブラ言語はドメインスペシフィックであり、一つのアセンブラニーモニックは一つの基本的なボイスメニュー動作に相当する。それゆえモデリングコンセプトとニーモニックのマッピングはシンプルで明確。

ジェネレータはリレーションシップに従って動作のフローを追いかけて、各デザインエレメント内の情報を変換されるコード内に出力する。例えば、モデル図左上の最初の吹出しから、1~5行目のSayコマンドを生成する。

音声オーディオのサンプリングは多くのメモリを要するので、各言葉は一度だけストアされ完全なセンテンスを構成するときに再利用される。それゆえメニューエレメント内の音声メッセージは個別の言葉、あるいは短いセンテンスのシーケンスと、そのオーディオサンプルに相当するメモリアドレスと合わせて構成される。

コードジェネレータはそのような構造に対して、単純に収集を繰り返して、各言葉にSayコマンドをサンプルのアドレスに応じて出力する。同じ音声が複数回現れたなら、ジェネレータは同じメモリアドレスに出力する。例えば4と10行目は同じメモリアドレスを使用している。

Sayコマンドのバリエーションは変数値（6や15行目）、あるいは引数入力（17行目）。Sayコマンドに加えて、変数値の設定（11行目）の動作も有る。

11~20行目に少しだけ複雑なジェネレーションプロセスを見ることができる。事前設定されたライフスタイル設定に対して選択値を読み上げる

```
11  Load A  00
12 :10_663
13  Add A  01
```

```
14  Say  0x09  1   'for...'
15  SayMode A
16  Say  0x00  1   '...press...'
17  SayNumber A
18  Test A <  05
19  If
20  Jump 10_663
```

最初に変数 A の選択を初期化（11 と 13 行目）。そして事前設定されたライフスタイルの情報を取り出してスピークする。（14~16 行目、ただしスペースの都合で簡素化している）そして選択オプション（17 行目）。18~20 行目は条件分岐。もし選択変数 A が 5(05 hex)以下ならコードは 12 行目へジャンプバックして次の事前設定された Say モードへ、A が 5 になるまでループを繰り返す。

この事例でコード生成のキーはモデリング言語。ターゲット言語にはドメインスペシフィックなコマンドはわずかであり、ジェネレータ構築に便利なフレームワークがプラットフォームとして用意されて無い。また見ての通り、ジェネレーション処理に特別なトリックはない。モデルレベルでシステムの構造と振舞いのエッセンスをカバーしているので、ジェネレータはローレベルの言語に関わる複雑なバリエーションへの対処や実装上の課題に対処する必要がないので。

Example 3: Generating Python for Mobile Phones

コネクションに従ってモデル内をナビゲートすることは、モデルからコードに変換する方法の一つ。ここでは個別のモデルエレメントごとに関数定義を生成する例として、スマートフォンのアプリを生成させる例を紹介する。S60 のプラットフォームと Python フレームワークが存在して、一連の API を提供し、ユーザインターフェイスに対する特定のプログラミングモデルを要求する。モデルベースの生成を可能にするために、仕様化言語（DSM）とジェネレータはプログラミングモデルと API に従う。Fig 6 は、このサンプルモデル。

Fig 6 内のモデリングコンセプトは、S60 スマートフォンのアプリ開発用のサービスやウィジットを直接ベースにしている。モデリング担当者はアプリの振舞いのロジックを、ウィジットの振舞いや実製品で提供されるアクションをベースにして記述。もし携帯電話のアプリ（アドレス帳やカレンダー）に慣れ親しんでいるなら、モデルを見ればどのようなアプリであるかを理解できる。

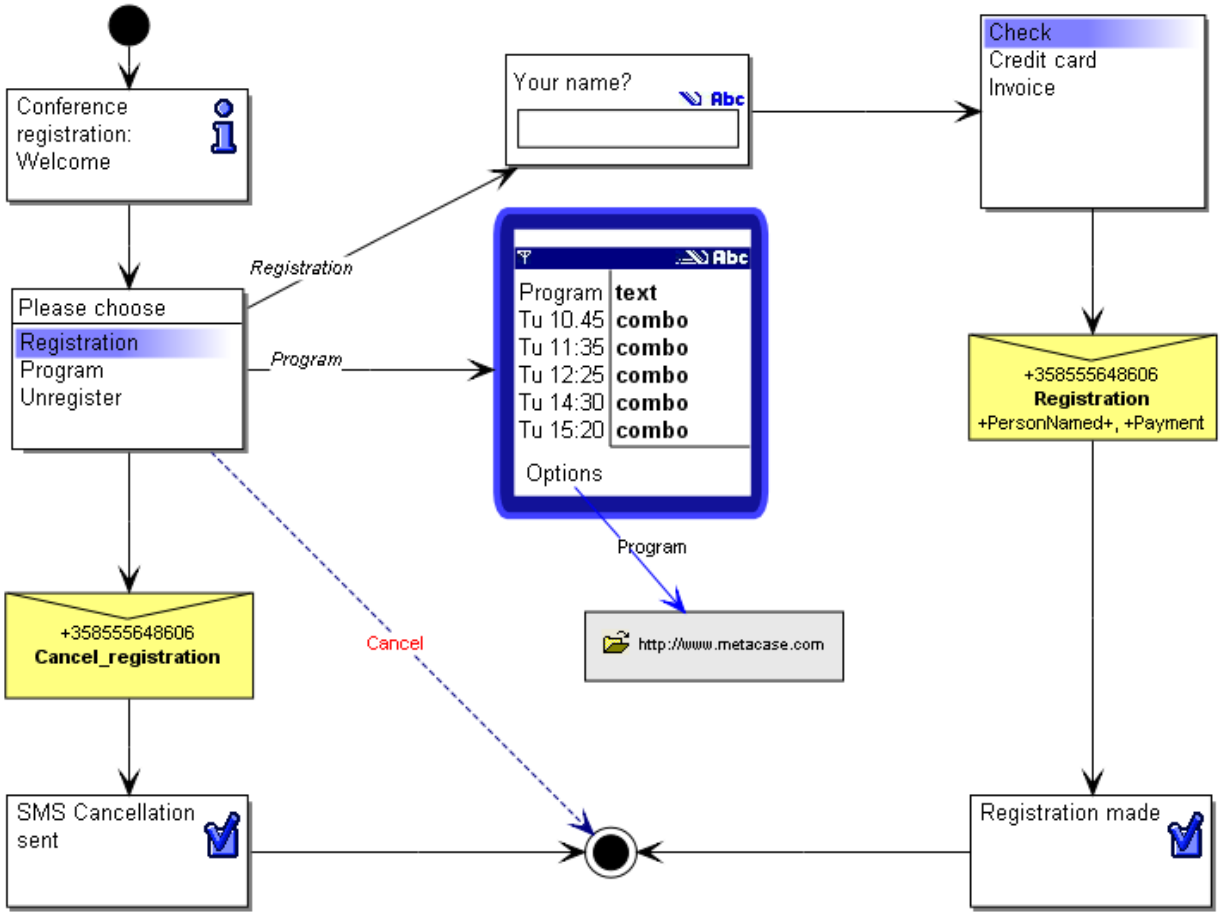

Fig 6. Conference Application: The figure shows the design for a conference application intended to run on a S60 phone.　S60 スマートフォン用のカンファレンス登録アプリ

ジェネレータは Fig 6 のデザインモデルから、コンパイルして実機やエミュレータで実行可能な関数ベースのコードを生成する。ジェネレータはモジュールに構造化される（各モデリングコンセプトに一つのジェネレータモジュール）。例えば一つのジェネレータモジュールがリストを扱って、他のモジュールが承認ダイアログを処理するとか。様々なコンセプトが同様のコードの生成を必要とするので（次のコンセプトへの制御フローなど）、ジェネレータ定義のパーツはサブルーチン化され、様々な箇所で使用される。ジェネレータはディスパッチや複数ビュー（表示窓上のタブなど）の管理用のいくつかのフレームワークコードを抱える。以下、Fig 6 から生成される Python コード例。

```
01  import appuifw
02  import messaging
03
04  # This app provides conference registration by SMS.
...
33  def List3_5396():
34  # List Check Credit card Invoice
35      global Payment
36      choices3_5396 = [u"Check", u"Credit card", u"Invoice"]
37      Payment = appuifw.selection_list(choices3_5396)
38      if Payment == None:
```

```
39          return Query3_1481
40      else:
41          return SendSMS3_677
...
85  def SendSMS3_677():
86  # Sending SMS Conference_registration
87  # Use of global variables
88      global PersonName
89      global Payment
90      string = u"Conference_registration "\
91            +unicode(str(PersonName))+",  "\
92            +unicode(str(Payment))
93      messaging.sms_send("4912345678", string)
94      return Note3_2227
...
101  def Stop3_983():
102  # This applications stops here
103     return appuifw.app.set_exit
...
107  f = Note3_2543
108  while True:
109     f = f()
```

ジェネレータは、使用されるサービスをベースに、モジュールのインポートを 1~2 行目に生成。先に UI アプリケーションフレームワークで、次にメッセージ用のモジュール（モデルに SMS アクションがあるので（黄色封筒））
インポートに続くコメント文は、デザイン上に記述されたドキュメントから取得。
次にコードは各サービスとウィジットを独自の関数として宣言。ジェネレータは不特定な順に関数を生成するのではなく、タイプごとに生成させる。例えば全てのリスト関数は SMS/text message 関数群に続く。

33~41 行目はリスト用のウィジットを使って支払方法を選択するコード。関数名とコメントを定義した後、コードはグローバルの Payment 変数を宣言する。36 行ではリストの値を Unicode でローカル変数へ。37 行目ではフレームワークで提供されるリスト用のウィジットをコール。85~94 行目でコードは SMS メッセージ送信を処理（リスト用のウィジットと同様の方法で）。93 行目はインポートされた SMS モジュールの sms_send 関数をコール。ジェネレータはモデルからパラメータ(recipient number, message keyword and content)を取って関数に渡し、正しいメッセージのシンタックスの形成を行う。これらメッセージは明確に定義され、いつも同じパターンを用いる。

各関数の終わりの部分に、ユーザ入力をベースに次の関数へのコールが含まれる。SMS 送信のために、ジェネレータはシンプルにアプリケーションフローに従うが（94 行目）、リスト選択のために、少々複雑になる。リストからのユーザ選択によって、異なる選択が存在する。この場合リストから選択される値に関わらず、コードはいつも同じ処理を取るが、操作のキャンセルも考慮に入れる必要がある（Cancel か Back ボタンで）。ジェネレータは前のウィジットに実行を戻すキャンセル動作のコード（38，39 行目）を自動生成する。もし選択されると SMS 送信（40~41 行目）に進む。キャンセルあるいは例外のアクションを DSM 内に暗黙的に持たせれば、アプリケーションのモデリングがよりシンプ

ルになる。言い換えると、モデリング担当者は大抵の場合何もする必要なく、初期状態と異なる振る舞いを指定するだけで良いようになる（最初のメニューからエンドステート：黒二重丸への対角線で引かれた ------Cancel-----> （キャンセル）のリレーションシップなど）

最後の関数では、ジェネレータはアプリのエンドステートを基準にアプリケーションの終了コードを生成（101~103 行目）そして最後はディスパッチャーが最初の関数を呼んで（107 行目）アプリを開始。他と同様この関数は、コールするために次の関数をリターンする。そして 108~109 行目で各関数が終了したら次の関数コールを処理する。この方法でコールを処理することは、ある種の再起呼出しを提供するので別の機能に移るような場合のスタックの深さを軽減する。（各関数に次の関数をコールさせるよりも）

Example 4: Generating C for Digital Wristwatch Applications

最後の例はステートマシンから C ソースコードの生成を紹介。ステートマシンコードの生成には様々な方法があるが、プログラミング言語とプラットフォーム、コードのサイズなど状況による。

デジタル腕時計のサンプルのアーキテクチャでは、現在時刻、ストップウォッチ、アラームなどを表示する一連のアプリケーションで構成される。モデリング言語はボタン、時間の単位（時・分・秒）、アイコンといった時計アプリの静的なエレメントの定義と、ステートマシンを使用するような機能の定義に主眼を置く。そしてトラディッショナルな状態遷移図のセマンティクスをドメインスペシフィックなコンセプトや制約で拡張する。例えば、各ステートは時間を表示する機能にリンクする。状態遷移は一つのボタンを一度押すことでのみトリガされる。算術処理は時間の処理に相当する箇所に限られるなど。Fig 7 は、現在時刻の表示・編集、関連操作のサンプルモデル。そして生成される C コードの一部を紹介する。

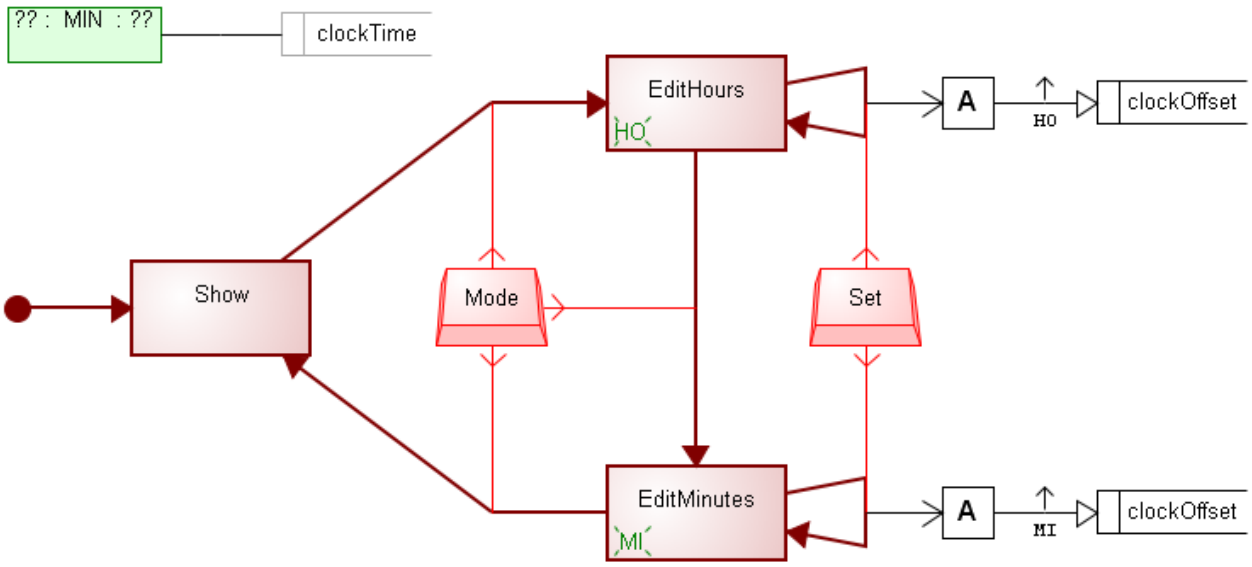

Fig 7. Watch Application Design: The figure shows the design for a watch application that can display and edit the current time.

```
01 typedef enum { Start, EditHours, EditMinutes, Show, Stop }
   States;
02 typedef enum { None, Mode, Set } Buttons;
03
04 int state = Start;
05 int button = None; /* pseudo-button for following buttonless
                        transitions */
06
07 void runWatch()
80 {
09   while (state != Stop)
10   {
11     handleEvent();
12     button = getButton(); /* waits for and returns next button
                                press */
13   }
14 }
15
16 void handleEvent()
17 {
18   int oldState = state;
19   switch (state)
20   {
21     case Start:
22       switch (button)
23       {
24         case None:
25           state = Show;
26           break;
27         default:
28           break;
29       }
30     case EditHours:
32       switch (button)
33       {
34         case Set:
35           roll(clockOffset, HOUR_OF_DAY, 1, displayTime());
36           break;
37         case Mode:
38           state = EditMinutes;
39           break;
40         default:
41           break;
42       }
43     case EditMinutes:
...
55     case Show:
...
64     default:
65       break;
66   }
```

```
67    button = None; /* follow transitions that don't
                         require buttons */
68    if (oldState != state) handleEvent();
69  }
```

モデルからコードへのマッピング作業は容易。ジェネレータはデザイン内からステートやボタンのユニークなラベルになる列挙子(enums)を生成（1~2 行目）。そして 4~5 行目で初期化。そして全アプリに共通の runWatch()関数の定型文を出力。各入力イベントに対し、runWatch()はアプリの振舞いの主要箇所である（handleEvent() (lines 16—66)）を呼び出す。handleEvent() では、次に何を何処に行ってするかは先立つステートと、入力イベントが何であったかに依存する。ジェネレータはこのようにして状態遷移図を読み、単純なネストのスイッチ文で実装する。ジェネレータはこれをモデル内のステートに対して繰り返して生成する。各ステートが終了するまでの繰り返しの遷移を。

トランジションはアクションをトリガすることもできる。このモデル例では算術操作は時間単位のサポートのみ。時間変数の設定は単純。しかし時、分それぞれを上下に調整するときに繰り上がりなどの考慮が必要。このような処理は、他の機能でも必要なので、フレームワークのコンポーネントにしたほうが良い。そして必要な箇所でそのコールのみを生成させるようにすれば（35 行目など）、コードはコンパクトでスマートにできる。