

2002年度  
卒業論文

UMLを用いた分散システム・  
モデリングツール

神戸大学工学部情報知能工学科  
森 健司

指導教官 林晋

2003年2月21日

# UMLを用いた分散システム・モデリングツール

森 健司

## 要旨

ハードウェアの進歩によって、組み込み機器に搭載されるソフトウェアの大規模化が進んでおり、これに対応した新しい開発技法が求められている。林研究室では SMART というツールを開発することを通じてこの問題への解決を図ろうとしている。

SMART では、UML Statechart 図を用いたシステムの設計が可能であり、設計したシステムのモデルを動作させることができる。さらに、インターフェースの構築環境も提供しており、モデル動作時に Statechart 図で表現されたシステムの動作モデルに対してイベントを送ることができる。

この SMART を情報家電のモデリングに対応させるために、林研究室の玉川世宗、花山健二と共同研究を行っている。そのうち本研究では SMART が複数のアプライアンスのモデルを同時に扱えるように、昨年度に作成された SMART0.1 を基に、SMART0.2 を実装した。

SMART0.2 においては設計対象のシステムのユーザーインターフェースを View Point という単位で定義できるようにした。View Point はシステムの見た目をあらわすもので、一つのオブジェクトに対して複数の View Point を設定することもできるし、複数のオブジェクトをまとめて一つの View Point で表現することもできる。ただし、SMART0.2 に実装した View Point は、View Point の追加・変更を行うたびにオブジェクトのモデルを変更しなければならず、ソフトウェア開発には不十分である。そこで、より柔軟な開発を行えるようにするために、View Point の変更・追加を行っても設計対象オブジェクトの動作モデルに変更を及ぼさないように改善された View Point の概念を提唱した。

# 目次

第1章 序論	4
第2章 UML	6
2.1 Statechart 図	6
2.1.1 状態 (State)	6
2.1.2 状態遷移 (Transition)	7
2.1.3 入れ子構造の状態 (Composite State)	8
2.1.4 最終状態 (Final State)	9
2.1.5 擬似状態 (PseudoState)	9
2.1.6 同期状態 (Synch State)	11
2.2 属性 (Attribute)	11
2.3 Action Semantics	13
第3章 SMART	14
3.1 概要	14
3.2 ツールの実装	14
3.2.1 User Interface Modeler	15
3.2.2 Statechart Modeler	17
3.2.3 Action Semantics に基づいたミニ言語	18
3.3 論理モデルと外見モデルの切り分け	19
3.4 ミニ言語を使った通信について	20
3.4.1 Statechart 図上で行われる通信	20
3.4.2 Statechart 図から View Point への通信	21
3.4.3 View Point から Statechart 図への通信	21
3.4.4 View Point 上で行われる通信	22
第4章 View Point の拡張	24
4.1 SMART0.2 における View Point の問題点	24
4.2 改善された View Point の概念	25

<b>第 5 章</b>	<b>既存ツールと SMART との比較</b>	<b>27</b>
5.1	IIOS . . . . .	27
5.2	Rational Rose RealTime . . . . .	28
<b>第 6 章</b>	<b>結論および今後の展開</b>	<b>29</b>
6.1	結論 . . . . .	29
6.2	今後の展開 . . . . .	29
6.2.1	完全な View Point の実装 . . . . .	29
6.2.2	Action Semantics に基づいた並列動作可能なエンジン . . . . .	29
6.2.3	階層モデルのサポート . . . . .	30
6.2.4	UML Class 図のサポート . . . . .	30
6.2.5	時間モデルの導入 . . . . .	30
6.2.6	モデルの検証 . . . . .	30
	<b>謝辞</b>	<b>32</b>
	<b>参考文献</b>	<b>33</b>

# 第1章 序論

近年のハードウェアの進歩によって、組み込み機器の分野では規模の大きなソフトウェアに対応した新たな開発技法が求められている。この問題を解決するため、林研究室ではオブジェクト指向組み込みソフトウェア開発用の SMART というツールを作成している。SMART ではシステムの設計・動作検証が可能であり、主な特徴として以下のものが挙げられる。

- UML Statechart 図を使ったシステムの動作設計
- 物理インターフェースの作成
- Action Semantics に基づいた平易な言語による動作記述の統一

しかし現在の SMART には複数機器の分散実行ができないという欠点が存在する。このため、SMART の分散システム対応化を行っている。本研究はこの分散システム対応化の研究の一環であり、同研究室の玉川世宗および花山健二の研究との共同研究である。

本研究の基となった SMART システムは、昨年度に林研究室の谷内上智春が作成した。この古い SMART を SMART0.1 と呼ぶ。

本研究では SMART0.1 を複数機器のモデルを同時に扱い、それらのモデル同士の通信が行えるように改良した SMART0.2 の実装を行った。そのためにインターフェース設計用に View Point という概念を考案した。View Point とはシステムをユーザから見る時の視点であり、シミュレーション実行時にはこの View Point からユーザの入力を受け取り、モデルの動作を実行する。一つのオブジェクトに対して複数の View Point を定義したり、また複数のオブジェクトをまとめて一つの View Point の中で表現することができ、柔軟なシミュレーション実行環境が容易に構築できる。

しかし、SMART0.2 に実装した View Point の概念では、システムの動作モデルが View Point に依存しており、新しく View Point を定義するたびにシステムの動作モデルを変更する必要があるので、ソフトウェア設計に使用するには不十分である。そこでシステムの動作モデルが View Point に依存しないように、View Point の概念を改善したものを提唱した。

共同研究者の玉川は、現在情報家電の先駆的存在である Echonet 規格に基づいた、EMS(Energy Management System) のシミュレータを作成し、SMART がより一層情報家電のモデルに適合するためにどのような機能が必要になってくるかを探っている。また、同じく共同研究者の花山の研究は SMART の中核となっている Action Semantics に基づいたエンジンの並列動作化をすすめている。これらの研究の成果は将来的に統合され、現行の SMART のようなツールを作成するためのライブラリ群として SMART パッケージを構築するための布石となる。

本論文は本章を含めて 6 章から構成される。第 2 章では SMART の概念的な基盤となっている UML の概要について説明する。第 3 章では従来の SMART の問題点を踏まえた上で、SMART を新たにどのように実装していったかについて述べる。第 4 章では 3 章での実装で使ったオブジェクトの見た目を表現する View Point の概念について問題点を述べた上で、改善した概念を提案する。第 5 章では SMART と類似した目的を持つツールをいくつか紹介し、それぞれの特徴および SMART との違いを述べる。第 6 章では本研究の成果および問題点、今後の方針について述べることによって本論文の結びとする。

## 第2章 UML

本章では、SMARTの基盤となっているUMLの概要について述べる。

UML(Unified Modeling Language)はオブジェクト指向のシステム開発において開発者同士の意志疎通をとりやすくするためのもので、図を使ったグラフィカルな表現のモデリング言語である。UML図には構造図としてClass図、Object図、Use Case図、Component図、Deployment図があり、振る舞い図としてStatechart図、Activity図、Sequence図、Collaboration図がある。

以降では本研究の実装で用いられているStatechart図、属性(Class図の概念)および現在策定中のAction Semanticsについて説明する。

### 2.1 Statechart図

Statechart図は、外部からの刺激に対してどのように応答をするかを定義することによって実体の動的な振る舞いを表現する図であり、状態や状態遷移などから構成されている。Statechart図で表現できるものはクラスのインスタンスだけではなく、ユースケース、アクター、サブシステム、オペレーションやメソッドといったものもこの図で表現することができる。本節ではStatechart図の主な構成要素について説明していく。

#### 2.1.1 状態(State)

状態とはオブジェクトの生存期間内における状況のことであるが、ここで言う状況には、何らかの条件を満たす状況、何らかの動作を行っている状況、何らかのイベントを待っている状況などが挙げられる。それぞれの状態には名前を付けることができる。オブジェクトが特定の状態であるとき、その状態はactiveであるという。状態は通常、角の丸い四角形で表現する。

状態には遷移の制御と連動して動作するアクションを関連付けることができる。状態と関連付けることのできるアクションとしてentry action、exit action、do-activityの三種類が存在する。状態がactiveになる直前にentry actionとして定義

されているアクションが実行され、同様に状態が active から抜け出す直前に exit action として定義されているアクションが実行される。状態が active である間中、do-activity として定義されているアクションは繰り返し実行され続ける。

また状態は internal transition を持つことができる。internal transition は状態遷移(後述)の一種であるが、発火しても状態の遷移が起きない。イベントに対して状態の変化なしにアクションを起こしたい時に使われる。

Fig 2.1 に、State の記述の仕方を示す。青色で描かれた部分は補足的な説明であり、本来の Statechart 図の構成要素ではない。これ以降の図においても同様である。

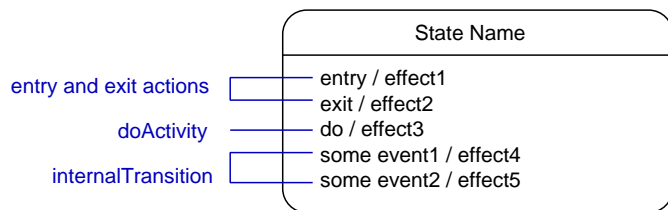


Fig 2.1: State

## 2.1.2 状態遷移 (Transition)

状態遷移は、二つの状態の関係をあらわすもので、状態から状態への移り変わりを意味する。要素として状態遷移を引き起こすきっかけとなるイベントを定義する trigger、状態遷移の起きる条件を定義する guard、遷移時に起こすアクションを定義する effect、遷移元の状態を定義する source、遷移先の状態を定義する target がある。Statechart 図では状態遷移は Fig 2.2 のように source 状態から target 状態への矢印で表現される。

遷移元と遷移先の状態が同一である状態遷移のことを特に SelfTransition(自己遷移)という。前述した internal transition と SelfTransition との違いとして、internal transition の場合は状態の entry/exit action が実行されないが、SelfTransition の場合だと実行されるということが挙げられる。

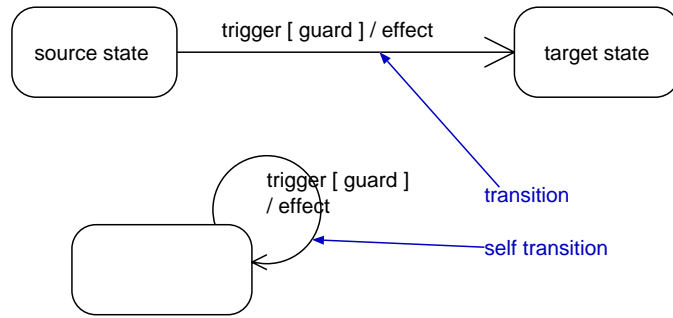


Fig 2.2: Transition

### 2.1.3 入れ子構造の状態 (Composite State)

内部に子状態を持つ状態を Composite State と呼ぶ。Composite State が持つ子状態には、排他的で互いに素な Sequential Substate と同時並列で実行される Concurrent Substate ( Region と呼ぶ ) の二種類がある ( Fig 2.3 )。Concurrent Substate を子状態として持つ親状態のことを Concurrent State という。Concurrent State は子状態に 2 つ以上の Concurrent Substate のみを持つことができる。

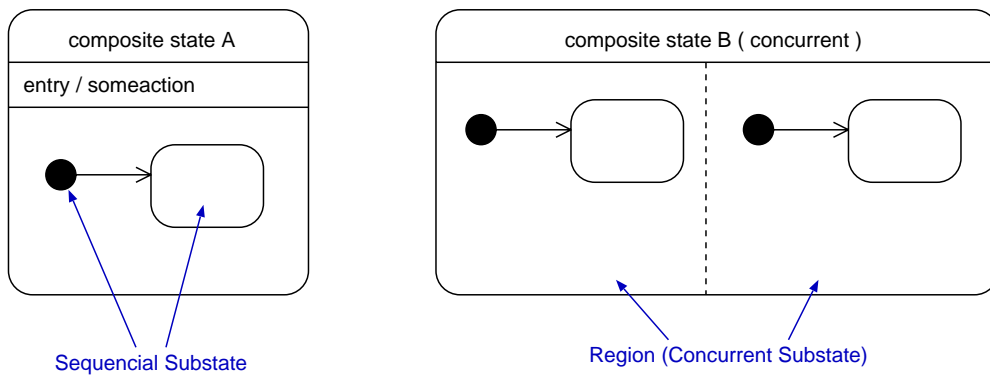


Fig 2.3: Composite State

## 2.1.4 最終状態 (Final State)

最終状態は特殊な種類の状態である。Statechart 図では黒い円に白い円を囲ったもので表現される (Fig 2.4)。最終状態が Active になると、その最終状態が属している Statechart や Composite State は内部の状態遷移を完了したという意味になる。最終状態は他の状態への状態遷移を持たない。



Fig 2.4: Final State

## 2.1.5 擬似状態 (PseudoState)

擬似状態は Statechart 図において一時的な遷移の制御の通り道になるもので、厳密には状態ではないが状態の一種として扱うことができる。上記の理由で擬似状態は Active にはならない。擬似状態の種類として初期状態、履歴状態、結合状態、分岐状態があり、それぞれの意味は以下の通りである。

### 初期状態 (Initial State)

Initial State は遷移の開始地点を表すものである。Statechart 図では黒丸で表現される (Fig 2.5)。Composite State が Active になった際、その Composite State の子状態のうち Initial State から遷移で結ばれた状態がデフォルトとして Active になる。同じ Composite State 内に Initial State を 2 つ以上定義することはできない。

Initial State は正式な状態とは違い Active にはならないので、必ず Trigger となるイベントのない他の State へと向かう遷移を定義しなければならない。また、Initial State に向かう遷移を定義してはならない。そういった遷移はその Initial State の親となる Composite State に対して接続する。

### 履歴状態 (History State)

History State は、自分が属する Composite State 内の子状態や孫状態のうち、どれが Active であったかを記憶し、それを復元することができる。記憶が行われる



Fig 2.5: Initial State

のは親となる Composite State が遷移によって Active でなくなる瞬間である。また記憶したものが復元されるのは History State に遷移が移った時である。

History State は記憶できる範囲の違いでさらに Shallow History と Deep History の二種類に分類される。Shallow History が記憶できる範囲は、ShallowHistory と同じ Composite State を親とする State だけであるのに対して、DeepHistory はさらにその子状態、孫状態といった風に全ての子孫状態が記憶の範囲となる。Shallow History は Statechart 図では “ H ” という文字を囲った円で表現される。Deep History の場合は “ H ” という文字の横に “ \* ” という文字を付ける (Fig 2.6)。



Fig 2.6: History State

### 結合状態 (Join Vertex)

Join Vertex は同じ Composite State 内の別々の Region からの状態遷移を結合させる役割を持つ擬似状態である。Statechart 図では太い線を使って表現される (Fig 2.7)。2 つ以上の状態遷移の入力および 1 つの状態遷移の出力を定義する。入力側の状態遷移が全て発火した時に出力側の状態遷移が発火する。

### 分岐状態 (Fork Vertex)

Fork Vertex は遷移の制御を分割する役割を持つ擬似状態である。Statechart 図では Join と同様に太い棒を使って表現される (Fig 2.8)。1 つの状態遷移の入力と 2 つ以上の状態遷移の出力を定義する。入力側の状態遷移が発火した時、出力側の

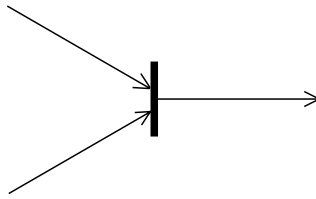


Fig 2.7: Join Vertex

状態遷移は全て発火する。

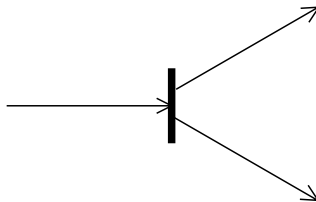


Fig 2.8: Fork Vertex

### 2.1.6 同期状態 (Synch State)

Synch State は同じ Concurrent State 内の複数の Region 内の遷移の同期を取る特殊な状態である。Fork Vertex や Join Vertex と一緒に使う。要素として bound を持つ。bound は Synch State の入力側の遷移と出力側の遷移の発火数の差の最大限度値であり、正の整数もしくは unlimited が指定できる。。Statechart 図では bound の値 (unlimited の場合は “ \* ” を使う) を囲んだ丸い円で表現される (Fig 2.9)。

## 2.2 属性 (Attribute)

UML にはオブジェクトが保持する型や値を定義するための属性という概念がある。Statechart 図だけでオブジェクトが保持する値 (例えば音響機器の音声出力

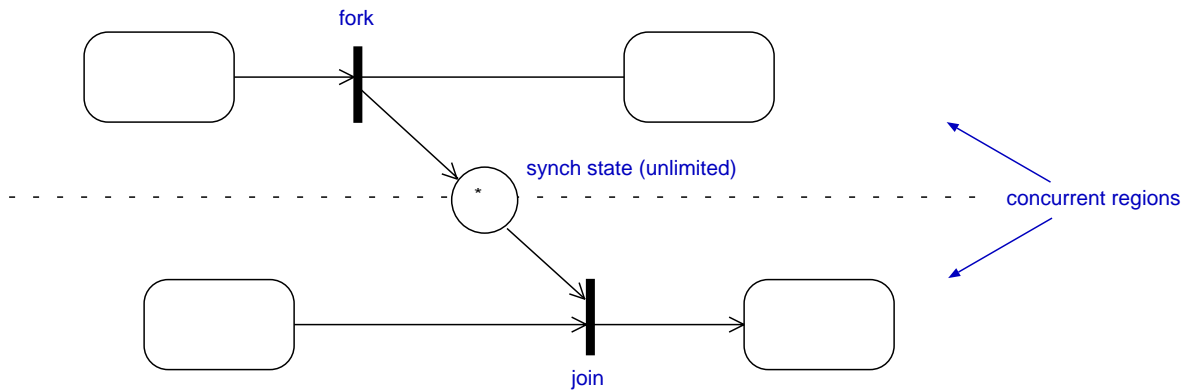


Fig 2.9: Synch State

ボリュームなど)だけで表現することは困難であるので、本研究のシミュレーションツールを作成するにあたり、これを導入している。属性の文法は以下の通りである。

```
visibility name : type-expression [ multiplicity ordering ]
    = initial-value { property-string }
```

それぞれの要素の意味は以下の通りである。

**visibility(可視性)** 他のオブジェクトからの見え方をあらわす。public、protected、private、.package の四種類が存在する。なお、それぞれの意味は定義されていず、使用者が自由に設定できる。

**name(識別名)** 属性を特定するための名前。

**type-expression(型)** 属性の型をあらわすためのクラス名やデータ型。

**multiplicity(多重度)** 属性の集合の度合いをあらわす。“最小値.. 最大値”の書式を取る。

**ordering(順序付け)** 属性が集合である場合、順序付けがどうかを表す。ordered と unordered の二種類があり、ordered が順序付けあり、unordered が順序付けなしをあらわす。省略した際は unordered となる。

**initial-value(初期値)** 属性の初期値。

property-string(プロパティ) 属性の変更可能性など、上記以外のものはここで記述する。

## 2.3 Action Semantics

UML モデルの動作の意味づけは Action Semantics で行われている。現在はまだ策定中で、UML にはバージョン 2.0 で取り込まれることになっている。

Action Semantics では UML モデルを動作させるための Action のモデルが定義されている。UML モデリングツールのベンダーのためにあるもので、多くの部分が Undefined Semantics としてわざと定義されておらず、ツールごとに自由に実装することが許されている。例えばスケジューリング方法などは指定しておらず、極端な話だと逐次動作させることすら許されている。

action の実行モデルはデータフローマシンによく似ている。それぞれの Action は Pin というものを持っており、処理対象のデータが流れる DataFlow というパイプを pin に接続することによって他の Action と接続していく。pin には InputPin と OutputPin があり、DataFlow は Action の InputPin と OutputPin につながる。

複数の action を実行する順序に制約がある場合は ControlFlow と呼ばれるものでそれを表現することができる。

本研究で実装したシミュレーションツールには、オブジェクトの通信にこの Action Semantics に基づいた独自実装の逐次実行型のスクリプト言語を使用している。

# 第3章 SMART

## 3.1 概要

SMART(Statechart Modeling and Action Requesting Toolkits) は林研究室の卒業生である谷内上智春が 2001 年度卒業研究用に開発した Statechart 図を用いた組み込みシステム設計支援ツールである。本研究ではこの一部をもとにして、情報家電のモデルを本ツールでシミュレートできることを目標として新たな実装を行った。以降、従来の SMART を SMART0.1 と呼び、本研究で実装したツールは SMART0.2 と呼ぶことにする。

## 3.2 ツールの実装

SMART0.1 の問題点として、以下のようなものが存在した。

1. 複数機器のモデルが扱えない。ツール上で作成&実行できるモデルは一つまでである。
2. イベントに対するアクションの動作が逐次実行であり、並列実行ができない。
3. 作成できる物理インターフェースのデザインが Java Swing に依存したもので自由性が少ない

2については、花山が並列実行が可能な Action Semantics の実行系の研究を進めていき、この並列実行可能なアクション実行エンジンが SMART0.2 に搭載される予定である。

本研究では、主に 1 と 3 の問題を解決した。

情報家電のモデルのシミュレーションを行えるようにするため、複数機器のモデルを扱うことができ、またそれらがお互い自由に通信できるようにした。

さらに、インターフェース設計において、ユーザが用意する画像ファイルを部品の外観として使用できるようにするなどによって、自由なデザインの設計を可能にした。

SMART0.2 は主に以下の構成で成っている。

User Interface Modeler 物理インターフェースの作成および実行。

Statechart Modeler オブジェクトの動作を Statechart 図でモデリング&実行。

Action Semantics に基づいたミニ言語 モデル間の通信役となる。

本節ではこれらについて詳細に説明をしていく。

### 3.2.1 User Interface Modeler

User Interface Modeler ではユーザがオブジェクトの操作をするためのユーザーインターフェースの作成および実行を行うことができる。SMART0.2 を起動させるとまずこの User Interface Modeler のウィンドウが画面に表示される (Fig 3.1)。組み込み機器の開発では、設計よりも前に仕様の中でインターフェースのデザインが与えられているケースが多いことを考慮してこのようにした。

家電などの組み込みシステムでは、ユーザーがシステムを操作するための User Interface はボタンや LED 表示などの単純なものでなければならない。最近のデジタル家電においても操作画面は GUI 化してきてはいるが、フォーカス管理処理 [Haru 02] を利用することによって、直接人間が触る User Interface 自体はボタンなどの単純なもののみで構成されたリモコンが主に使われている。家電などの組み込み機器の対象ユーザはコンピュータソフトウェアに慣れていない人にも及ぶので、キーボードやマウス操作などの複雑な動作が要求される User Interface はユーザから敬遠される。また、コンピュータソフトウェアに慣れている技術者であっても、日常的に使用する組み込み機器に対しては直感的で分かりやすい User Interface を好む傾向がある。この User Interface Modeler はこういった単純な User Interface を前提としている。

複数の機器を同時に扱えるようにするにあたり、View Point の概念を考案し、導入することにした。View Point とはオブジェクトを目で見る時の視点である。シミュレーションを実行するにあたり、モデルとユーザとのインタラクションをつかさどるオブジェクトの外観は一つだけでいいとは限らない。例えば現実のオブジェクトは 3D 構造であるのに対して SMART0.2 を実行するコンピュータの画面は 2D である。このため、オブジェクトを表現するのに都合のよい視点は一つとは限らず、上や横から見た図など、オブジェクトに応じてシミュレーションに都合のよい視点がいくつも存在する可能性がある。また、多数の家電が存在する部屋を見る時のように、複数のオブジェクトをまとめて見る視点というのも考えられる。

このツール上で View Point を複数作成することができ、その上に Statechart Modeler で構築するモデルに対して状態遷移を発火させるイベントを起こすことができる User Interface の部品を配置させることができる。Action Semantics に基

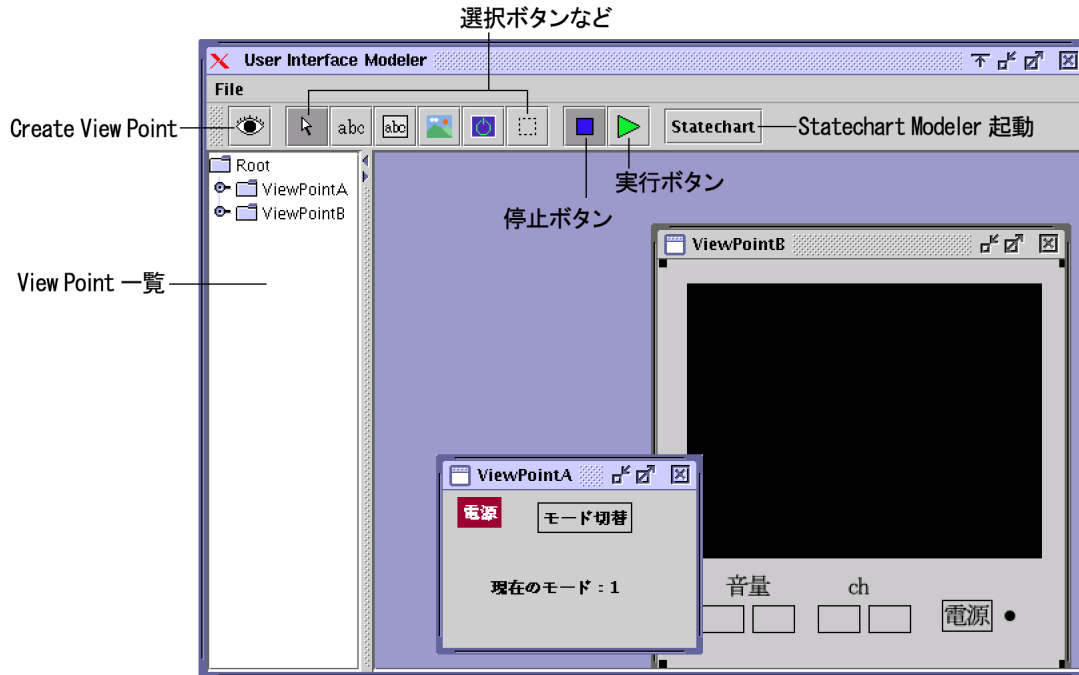


Fig 3.1: User Interface Modeler

づいたミニ言語で記述されたスクリプトを部品のアクションとして定義することによって、シミュレーション実行時においてユーザからの刺激を受け付けた瞬間に様々な動作を行わせることができる。また、部品の見た目として、色を自由に指定したり、ユーザが用意する画像を使用したりすることができ、自由なデザインで View Point を構築することができる。

このツールには二つのモードがある。一つは設計を行う Modeling Mode、もう一つがシミュレーションを実行する Executing Mode である。起動時は Modeling Mode であるが、実行ボタンを押すことによって Executing Mode に移行し、シミュレーションを開始することができる。停止ボタンを押すとシミュレーションが終了し、Executing Mode から Modeling Mode へと戻ることができる。

Execution Mode に入ると View Point 上に配置した部品は User Interface としての動作を行う。例えばボタンの場合は、クリックされるとそのボタンのアクションとして定義されていたミニ言語スクリプトがコンパイル&実行される。後述す

る Statechart Modeler でオブジェクトのモデリングをしながらシミュレーションの実行を繰り返すことで、プロトタイプモデルによる開発が容易に行える。

### 3.2.2 Statechart Modeler

StatechartModeler はオブジェクトの動きを UML Statechart 図を用いてモデリングできる (Fig 3.2)。また、この Statechart Modeler では属性も定義できる (Fig 3.3)。

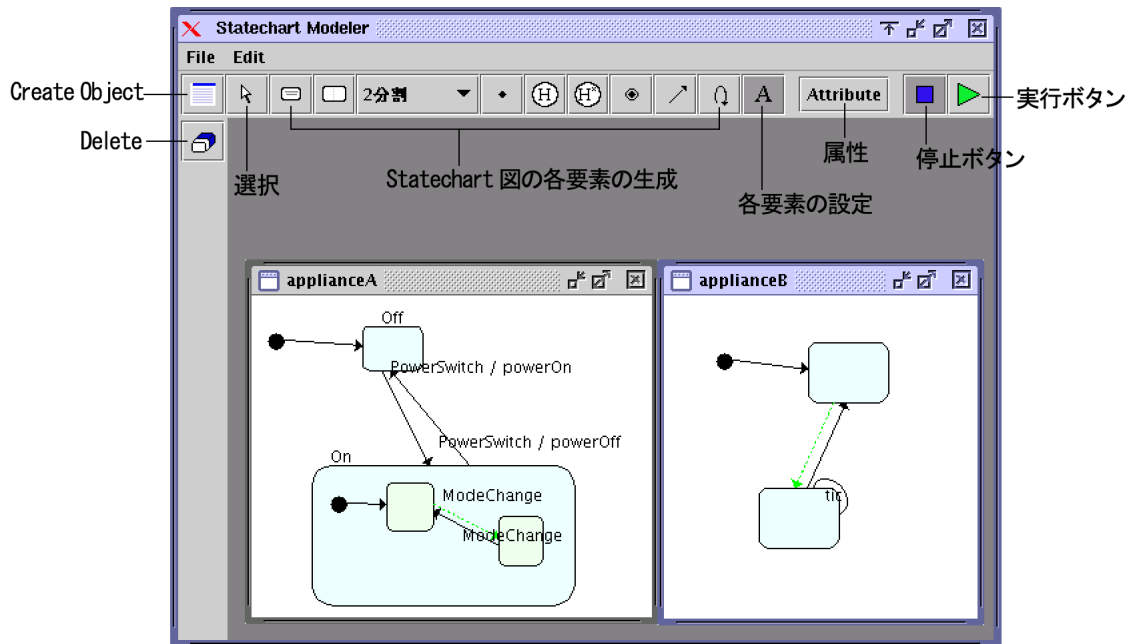


Fig 3.2: Statechart Modeler

ツールバーの一番左のボタンをクリックするとフレームが立ち上がり、その上で Statechart 図を作成できる。このフレームがオブジェクトに相当する。ツールバー上の各ボタンを使用することにより、Statechart 図によるオブジェクトの設計は一般のドロツールと同じ感覚で行える。

User Interface Modeler と同様に、オブジェクトの設計を行う Modeling Mode とシミュレーションを実行する Executing Mode の二つのモードがある。実行ボタンと停止ボタンによりこの二つのモードを切り替えることができる。

Execution モードに入ると、active な状態は赤く表現され、状態遷移をクリック

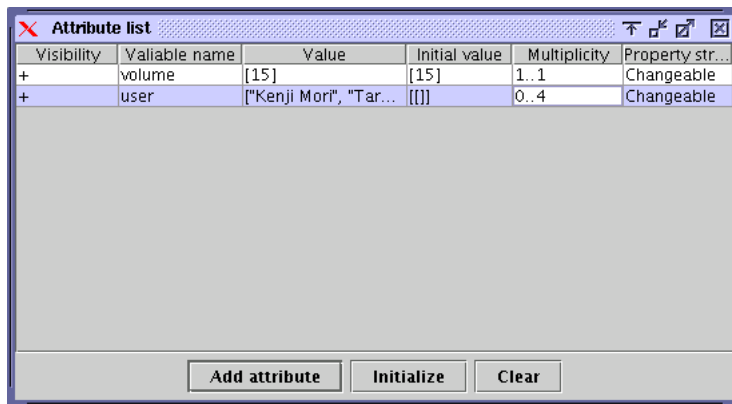


Fig 3.3: 属性の設定

することでその状態遷移を発火させることができる。イベント名の設定していない状態遷移については自動的に発火される。状態遷移が発火すると指定されたアクションが実行される。

### 3.2.3 Action Semantics に基づいたミニ言語

SMART0.2 では Statechart Modeler と User Interface Modeler 間のモデル実行の通信に独自実装のミニ言語が使用されており、Statechart Modeler の状態遷移や User Interface Modeler のボタンなどの部品アクションをこの言語で書かれたスクリプトで指定できる。この言語は Action Semantics に準拠している。主に以下のことが記述できる。

#### 基本的な演算子

四則演算 (“+”, “-”, “\*”, “/”) や代入 (“:=”)。

#### オブジェクトの属性の参照

文法：オブジェクトの識別名. 属性名

例：tv.volume := 5;

## View Point 上の部品のプロパティの参照

文法： View Point 名->サブ View Point 名->...->部品名. プロパティ名  
例: LivingRoom -> Television.image := ‘‘tvpoweroff.gif’’;

## トリガーイベントの発行

文法： ~‘‘オブジェクト名. イベント名’’  
例: ~‘‘tv.powerEvent’’;

### 3.3 論理モデルと外見モデルの切り分け

SMART0.2は将来的に SMART0.2と同じようなツールを作るためのライブラリ群にすることを想定している。そのライブラリを SMART パッケージと呼ぶ。現在実装されている SMART0.2は、改良を重ねた後、SMART パッケージを使ってできるツールのサンプルとして提供することになる。そのために Statechart Modeler や User Interface Modeler のモデルの実装を、論理的な意味しか持たないモジュールと開発者が作るツールに依存してしまう外見的な意味を持つモジュールとに分離するようにした。お互いは監視しあい、相手に関係する変化のあった際には相手に対して報告を行う。このように実装することで、開発者は SMART に含まれるモデル動作のエンジンをそのまま利用して、自分の設計しやすい独自のモデル構築&実行ツールを作成することができる (Fig 3.4)。

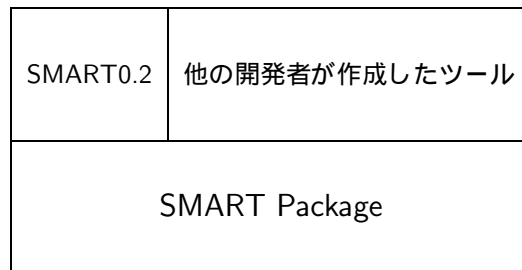


Fig 3.4: SMART Package

## 3.4 ミニ言語を使った通信について

SMART 上で Action Semantics に基づいたミニ言語を使用することによって可能となる通信の形態には四種類あり、状況に応じて適切に使い分けるべきである。本節ではこの四種類の通信の通信の形態および特徴について述べていく。

### 3.4.1 Statechart 図上で行われる通信

Statechart Modeler においてオブジェクトの Statechart 図の状態遷移に割り付けたアクションによって、別のオブジェクトの Statechart 図の状態遷移を発火させるトリガーイベントを発行することができる (Fig 3.5)。この通信は異なるオブジェクトに対するメッセージの役割を果たす。複数の機器間の通信が問われる情報家電においてはこの形の通信が重要なものとなる。

なお、同じオブジェクトの状態遷移を発火させる記述もできるが、このような記述を行うと Statechart 図を見ただけでシステムの動作を把握することが困難になってしまうので、Statechart 図中の状態遷移に展開するべきである。

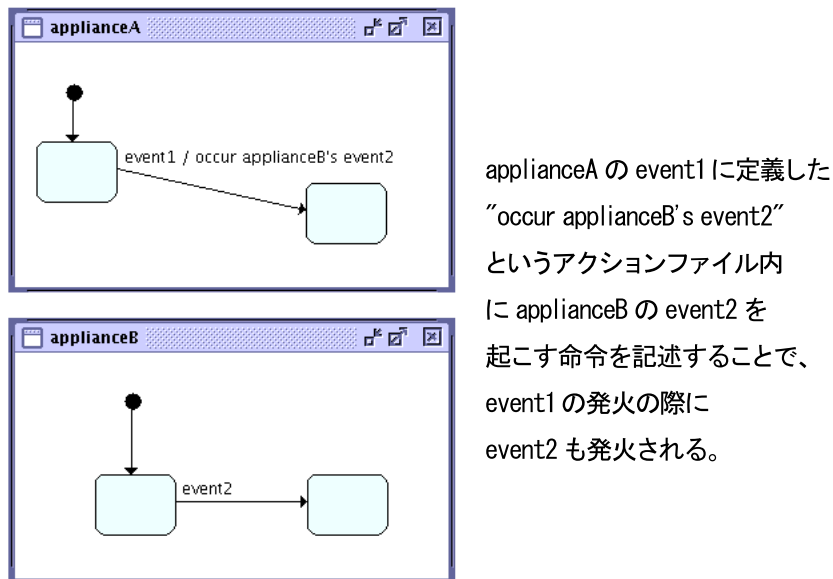
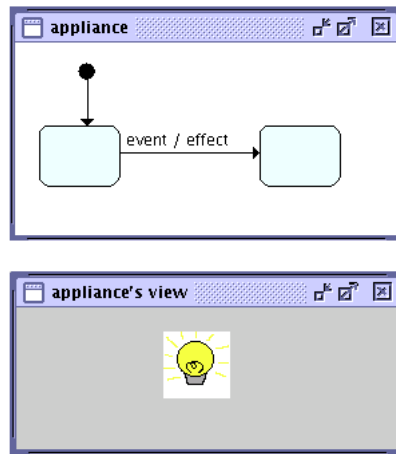


Fig 3.5: オブジェクト間の通信

### 3.4.2 Statechart 図から View Point への通信

Statechart Modeler においてオブジェクトの状態遷移に割り付けたアクションによって、User Interface Modeler 上の View Point に配置した任意の部品のプロパティを変更する通信を行うことができる (Fig 3.6)。この通信はオブジェクトの状態に合わせてオブジェクトの外観を変更する役割を果たす。しかしこの通信を記述するということは、オブジェクトのモデルが View Point に依存してしまうことになってしまうので、望ましくない。このことについて詳しくは次章で述べる。

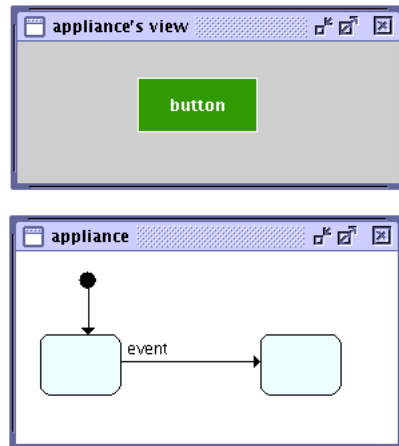


appliance の event に定義した  
"effect" というアクション  
ファイル内に appliance's view  
(View Point) の画像ラベルの  
画像ファイルを変更する命令を  
記述することで、  
event の発火と同時に画像が  
変更される。

Fig 3.6: オブジェクトの状態に合わせて外観の変更

### 3.4.3 View Point から Statechart 図への通信

User Interface Modeler において View Point の部品に割り付けたアクションによって、Statechart Modeler 上のオブジェクトの Statechart 図の状態遷移を発火させるトリガーイベントを発行することができる (Fig 3.7)。この通信はユーザの入力という外部刺激をオブジェクトのモデルへ通知する役割を果たす。

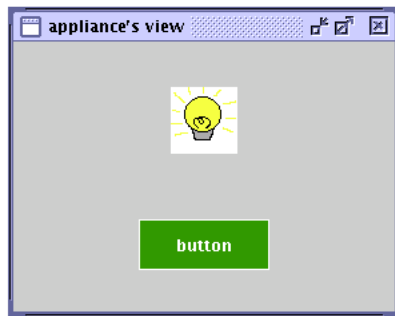


appliance's view (View Point) 上の button (テキストボタン部品) のアクションに appliance の event を発火させる命令を書いたアクションファイルを指定することにより、button をマウスでクリックした時、appliance の event が発火される。

Fig 3.7: ユーザからの入力のお知らせ

### 3.4.4 View Point 上で行われる通信

User Interface Modeler において View Point の部品に割り付けたアクションによって、他の部品 (他の View Point 上のものでよい) のプロパティを変更することができる (Fig 3.8)。しかし、オブジェクトの動作を表現しているはずの Statechart Modeler に対して何の断りもなしにオブジェクトの見た目だけが変更されるということは概念上あり得ないので、この通信は通常行うべきものではない。しかし便宜上オブジェクトの設計を簡略化するという意味で使うこともできる。これをうまく活用することにより、最終的なソフトウェアのプロトタイピングが容易に作成でき、そこから部分ごとに詳細化していくという形で柔軟な開発が可能となる。最終的にこのように記述された部分は完全に取り払わなくてはならず、自動的に検出できる機能が今後必要になってくる。



appliance's view (View Point) 上の  
button(テキストボタン部品)の  
アクションに画像ラベルの画像  
ファイルを変更する命令を  
記述したアクションファイルを  
定義することで、buttonを  
マウスでクリックした時に  
画像が変更される。

Fig 3.8: インターフェース同士の通信

## 第4章 View Pointの拡張

前章では User Interface の実装のために View Point の概念を述べたが、SMART0.2 に実装した View Point は単にオブジェクトの見た目というだけであり、ソフトウェアの設計に使うには十分ではない。本章では SMART0.2 上の View Point の問題点を述べ、その改善点について提案する。

### 4.1 SMART0.2における View Point の問題点

既に述べたとおり、View Point はオブジェクトを見るとき視点である。SMART0.2 では View Point 上の部品からオブジェクトの動作モデルにイベントのシグナルを送ったり、オブジェクトの動作モデルから View Point 上の部品に対して外観の変更を具体的に指示することができた (Fig 4.1)。しかしこれはオブジェクトの動作モデルが View Point に依存した形になっており、デザインや構成を変更した View Point を新たに定義する際などに、オブジェクトの動作モデルの定義もそれに応じて変更しなければならない。特に家電製品を開発する場合、製品の質がそのデザインに大きく影響されるため、様々なデザインによる検証を繰り返しながら最終的なデザインを決定することが望まれるので、見た目の変更は容易に行える必要がある。

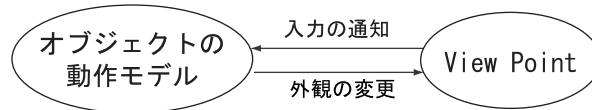


Fig 4.1: SMART0.2における View Point の問題点

## 4.2 改善された View Point の概念

上記の問題を解決するために、View Point を改めて以下のように定義する。まず、View Point とオブジェクト本体の動作モデルの中間に、User Interface のモデルを置く。User Interface のモデルは User Interface の外観に直接影響しない情報のみを要素として持つ。ここでいう外観に直接影響しない情報とは、例えば電灯の場合であれば点灯しているかどうかの boolean 値であるといった、その電灯が LED であっても蛍光灯であっても全く影響のない情報のことを指す。View Point はこの User Interface のモデルと関連付けられ、User Interface のモデルの状況に応じて外観を変更させる。一つの User Interface のモデルに対して複数の View Point を関連付けることもできる。

シミュレーション実行の際、本体の動作モデルは View Point に直接影響を与えることはできないが、その代わりにそれらの中間に定義された User Interface のモデルに対して直接的な影響力を持つ。User Interface モデルの情報が変更されると、最終的にその変更内容が View Point にも反映されることになるのだが、User Interface モデルから View Point 上の部品へとアクセスすることはできないことが注意事項として挙げられる。View Point が User Interface のモデルを常時監視し、User Interface のモデルの情報が変更されたことを検知し、自らの外観を変更させる (Fig 4.2)。またユーザからの入力は View Point 上の部品から部品モデル、本体モデルへと伝搬される。こうすることによって、本体の動作モデルや User Interface のモデルは View Point 上の部品について何も関知しなくてもいいので、View Point に全く依存しないものとなり、View Point の切り替えや追加が本体の動作モデルや User Interface のモデルの変更なしに行える。また、Fig 4.3 のようにイベントの受け渡しを行う仲介役を用意することで、現実世界にあるデバイスをインターフェースとして利用したシミュレーションも容易に実行できるようになる。

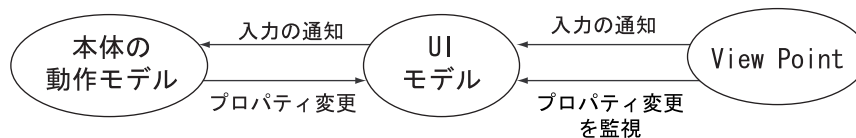


Fig 4.2: View Point

この提案は春名修介が [Haru 02] の中で述べているデジタル AV 機器向け GUI ソフトウェアの設計モデルと類似性を持つ。春名氏はソフトウェア開発フレームワークの MVC モデルをさらに二つに分離させ、デジタル AV 機器のリモコンで

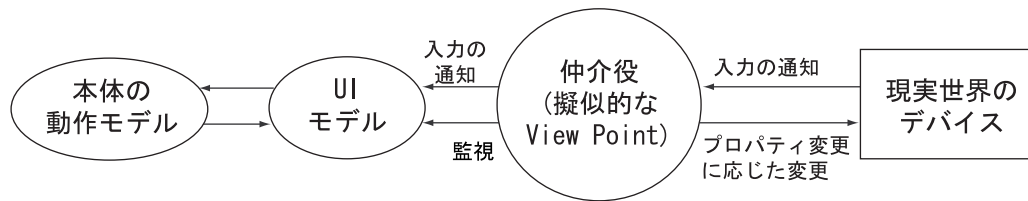


Fig 4.3: 現実世界のデバイスを利用したシミュレーションモデル

操作する GUI ソフトウェアのデザイン変更を容易に行えるようにしたと述べている。SMART0.2 で扱うオブジェクトを MVC モデルに当てはめると三つの役割はそれぞれオブジェクトの属性、Statechart 図で描かれるオブジェクトの動作モデル、View Point に相当する。今回の提案はこのうち View 部を View Point と User Interface のモデルとに分離させるということになり、この点で春名の述べたことと類似していると言える。

# 第5章 既存ツールとSMARTとの比較

本研究で実装したSMART0.2と同様に、UMLでモデリングしたオブジェクトを動作させることができるツールはいくつか存在する。本章ではそのうちIIOS、Rational Rose RealTimeについて紹介し、それらの特徴およびSMART0.2との比較について述べる。

## 5.1 IIOS

IIOS(Integrated Inter-exchangeable Object modeling and Simulation for Open-source software environment) はオープンソースで公開されているオブジェクト指向設計支援ツール、プロトタイピング開発環境およびコンポーネント構築支援ツールである。以下の6つの機能(ファシリティ)から構成されている。

MEF(Model Editing Facility) UMLモデル図を編集することができる。UML1.1に準拠しており、XMI形式での入出力が可能である。カリフォルニア大学アーバイン校で開発されたオープンソースUMLモデリングツールのArgoUMLが元になっている。

MDF(Model Debugging Facility) UMLモデルのシミュレーション/デバッグが行える。

IBF(Interface Building Facility) インターフェースの作成環境。

FCF(Format Conversion Facility) UMLモデルと他の表現形式との間の相互変換を行える。

DBF(DataBase Facility) 他のファシリティのデータを保存、更新することができるデータベース。

IDE(Integrated Development Environment) 全ての IIOSS ツールを統合する。ファイル管理、プロジェクト管理、各ファシリティの起動、ヘルプ機能などがサポートされている。

MDF では振り舞い図のシミュレーションやデバッグが行える。Statechart 図のシミュレーションが強力にサポートされていて、SMART の Statechart Modeler のシミュレーション実行と同様のことが行える。

IBF によって Java Swing を用いたの GUI の構築もサポートされているが、SMART のようにモデルに対してイベントを発行できるなどの今のところ実質的にモデルとの関連はない。

## 5.2 Rational Rose RealTime

Rational Rose RealTime は、Rational Software Corporation が製作・販売を行っている組み込みシステム向けソフトウェア設計ツールである。このツールは同社の Rational Rose を組み込みシステム向けに拡張したものである。このツールを使用することによって、UML を用いたモデル設計、動作および Java や C++ のソースコードを生成することができる。

モデル動作時には観測可能性インターフェースと呼ばれる外部プログラムをユーザが用意することによって、SMART と同様のシミュレーション実行が可能となる。観測可能性インターフェースは、TCP/IP ポートを使用して Rose RealTime 上のモデルと通信を行う。

しかし、この観測可能性インターフェースは、同社により通信用ライブラリが提供されてはいるが基本的にユーザが Java Applet 等で一から作り上げなければならず、構築が容易とは言い難い。

# 第6章 結論および今後の展開

## 6.1 結論

本研究の成果により、SMART0.2は分散機器に対応し、複数の機器のモデリングおよびシミュレーションの実行が自由に行えるようになった。また、SMART0.2の実装を通して、柔軟な物理インターフェースの設計に必要な View Point の概念を考案することができた。

## 6.2 今後の展開

### 6.2.1 完全な View Point の実装

第4章で述べた改良された View Point の概念をパターン化するなどして完成し、SMART0.2へ実装する。こうすることにより、設計対象のオブジェクトの外観を柔軟に切り替えることができ、実機を利用したシミュレーションにも対応できるようになる。

### 6.2.2 Action Semantics に基づいた並列動作可能なエンジン

SMART0.2では分散に対応したモデルを行うことが可能にはなったが、現段階でのシミュレーション実行はそれぞれのモデルが非同期に動作しているわけではない。分散機器のシミュレーションも本格的にサポートするためには、Action Semantics に基づいたアクション実行エンジンが並列実行可能である必要がある。

この研究は序章で述べた通り花山が行っている。SMART0.2は今後、その成果である Action Semantics に基づいた並列動作可能なアクション実行エンジンを搭載することになる。

### 6.2.3 階層モデルのサポート

SMART0.2 を情報家電に対応させるために必要なものとして、モデルを階層的に扱う機能が挙げられる。これは玉川の研究の成果であるが、Echonet の場合、通信方法の差異を吸収する目的で通信プロトコルに階層があり、上位の通信層でのやり取りは下位の通信層でのやり取りの上で行われる。これを SMART 上でモデルできるようにするには、Statechart 図に階層的な表現ができることが求められる。

### 6.2.4 UML Class 図のサポート

SMART0.2 においてはモデルの設計はオブジェクト単位になっているが、オブジェクト指向には欠かせないクラスの汎化や多態性を用いた抽象的な設計をサポートする必要があり、またオブジェクトのカプセル化を行うためにパッケージの概念も取り入れる必要がある。このために UML Class 図のサポートを行っていく。

しかし、SMART0.2 の特徴の一つとして、オブジェクト指向に精通していなくてもスムーズに設計を行える点が存在するので、これらの機能はオプションとして提供できるようにすることが望ましい。

### 6.2.5 時間モデルの導入

組み込み機器の分散環境においては時間という概念が欠かせないものとなる。特定の時刻に起きるイベントや、特定時間を待つイベントなどの時間イベントの概念モデルも取り入れる必要が出てくる。このモデルの導入によって、ビデオ録画予約や、一定時間毎に電力を監視するシステムなどのシミュレーションが構築できるようになる。

また、それぞれの機器が独自に動作するクロックを有することも考慮する必要がある。こうすることによってクロックがずれた機器同士を連携させるケースのデバッグなども行えるようになり、より高度な検証が可能となる。

### 6.2.6 モデルの検証

プログラム中にバグが潜んでいるように、ユーザが構築する動作モデル中にもバグが潜んでいる可能性は高い。そこでこのバグ早期的にを発見できるように、ユーザの構築したモデルが正しいものであるのかをチェックするモデル検証の機構を SMART0.2 へ実装する。制約としてボタンや LED などの User Interface の耐久性やフラッシュROM の書き換え限度回数などを設定することにより、通常のデバッ

グでは発見できないレベルのバグですら検出できるようにすることも視野に入れている。

# 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。

本研究は SMART project と松下電器産業との共同研究の一部として行われました。松下電器産業からの支援に感謝いたします。

また、SMART の実装および View Point の考案において多くの御指摘、アイデアをいただきました EIZO NANA O CORPORATION 谷内上智春氏、松下電器産業 春名修介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 潘沂冰氏および神戸大学工学部情報知能工学科 玉川世宗氏、花山健二氏に感謝いたします。

## 参考文献

- [UMLS 01] Object Management Group, “OMG Unified Modeling Language Specification”, <http://www.omg.org> (2001)
- [AUML 01] Object Management Group, “Action Semantics for the UML”, [http://www.kc.com/as\\_site/home.html](http://www.kc.com/as_site/home.html) (2001)
- [UMLU 99] Grady Booch, James Rumbaugh, Ivar Jacobson, “The Unified Modeling Language User Guide”, Addison-Wesley Pub Co (Sd) (1999)
- [UMLR 98] Grady Booch, James Rumbaugh, Ivar Jacobson, “The Unified Modeling Language Reference Manual”, Addison-Wesley Pub Co (Sd) (1998)
- [Alhi 98] Sinan Si Alhir, “UML IN A NUTSHELL: A Dektop Quick Reference”, O'Reilly & Associates (1998)
- [Suzu 01] 鈴木重徳, 倉骨彰, 佐野元之, 垣花一成, “IIOSS-UMLに基づく設計/開発環境のすべて”, 株式会社アスキー (2001)
- [Haru 02] 春名修介, “情報家電向け組み込みソフトウェア技術に関する研究”, 博士論文, 神戸大学 (2002)
- [Yach 02] 谷内上智春, “UMLを用いた組み込みシステムの開発支援系”, 卒業論文, 神戸大学 (2002)