修 士 論 文

# 拡張可能なモジュールをサポートするスクリプト言語

指導教官　中島達夫 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

則武 淳

1997 年 2 月 14 日

## 要旨

連続メディアアプリケーションの開発は、一般に困難であるために、これらを開発するための様々なツールキットが研究されており、多くのものは連続メディアを扱う部分をモジュールとして提供し、その組合せにより、アプリケーションを構成するという枠組を採用されている。しかし、従来の方式のモジュールは、モジュール化やその構造に問題があるために実際のアプリケーションの構築の際には、柔軟性や拡張性が犠牲になっていた。

本研究では、これを解決し、モジュールに高い独立性と再利用性を確保し、より柔軟なアプリケーションの開発が可能となる環境を構築するために、従来、行なわれていた連続メディアを処理するコード部分をモジュール化するだけではなく、それ以外の部分もモジュール化することを提案し、次いで、開発環境にイベントシステムを導入することにより、拡張可能なモジュールを構築するための新しいモジュールの階層化を提案する。

また、この手法の有効性を検証するためにこの開発環境をスクリプト言語とこのスクリプト言語により利用できるマルチメディアモジュールを実装した。スクリプト言語は、モジュールを生成し、メディアストリーム及びイベントを接続することで、アプリケーションを構築でき、リアルタイムシステムによる資源の管理が必要な動的 QOS(Quality of Service) 制御といったような低レベルな機能から、高レベルな GUI(Graphical User Interface) までをサポートすることで、拡張可能なモジュールをサポートした連続メディアアプリケーションの構築を可能としている。

この開発されたスクリプト言語により、容易に電子会議システムやビデオ・オン・デマンドシステムといった連続メディアアプリケーション開発環境の構築が可能となることを示す。

# 目 次

# 第 1 章

# はじめに

最近のパーソナルコンピュータの CPU を中心としたハードウェアの高性能化には著しいものがある。汎用のパーソナルコンピュータがビデオやオーディオといった連続メディアを、特別なハードウェアなしに扱うことは数年前まで困難であったが、CPU の駆動周波数の向上や、マルチメディア向けの命令セットやレジスタを備えるなど、CPU 自身の機能、性能が向上し、高いコストパフォーマンスにより低廉化している。そのため特にマルチメディアを標榜したものではない汎用機ですら、潜在的に連続メディアを扱える能力が整いつつあり、今後、これらの傾向はより一層進むことが予想される。

また、ネットワーク技術の進歩により、動画や音声などの連続メディアを含む、マルチメディアデータをネットワークを介して自由にやりとりすることも可能となっている。インターネットを介した、電話を模したアプリケーションや複数の地点から遠隔でマルチメディアアプリケーションを操作することができるようになっている [10, 9]。

このように、連続メディアを扱う環境は急速に整いつつある。

しかし、連続メディアの特徴として、処理に関する時間的制限が厳しく、情報量が膨大で、厳密なリソースの管理と実時間処理をサポートするシステムが必要となるため、アプリケーションビルダは、従来、オペレーティングシステムが行なっていた作業をアプリケーションに記述しなければならなくなる。

このような問題点を解決する一手法として、連続メディア処理の機能をモジュール化するものがある。これは、アプリケーションを連続メディア処理コード部分 (in-band code) とそれ以外の制御コード (out-of-band code) に分離し、あらかじめ in-band コード部を機能ごとにモジュールを作成しておくことで、アプリケーション・ビルダは、モジュールの構築と GUI の作成という out-of-band コード部の作成に専念できるという連続メディアアプリケーションの開発環境である [5]。

このような手法に基づき、分散ビデオプレーヤ Qtplay[7] のようなオペレーティングシステムの実時間処理や資源管理機構に強く依存した動的 QOS 制御を行なうアプリケーションを構築することのできるツールキットも開発されている [2]。さらにこのモジュールの制御をストリーム単位で行ない、ユーザにより開発しいやすいインタフェースを提供しているものとして、Cmt2 も研究・開発されている [17]。

このように多くの研究では、アプリケーションをin-band コード部と out-of-band コード部に分割し、そのうち in-band コード部をあらかじめモジュールとして提供することで、拡張性を確保することを目指している。

しかし、実際には、連続メディアアプリケーションを構築してみると、あるモジュールは特定のアプリケーションを構築する際にしか利用できないなどそれほど高い再利用性がない。

将来、ハードウェアやネットワーク環境の性能の向上によって、ますますユーザ・アプリケーションの巨大化や複雑化は、進むものと思われる。本研究では、従来の連続メディアアプリケーション開発環境におけるモジュール化の問題点を指摘し、将来予想される状況に対応するためのアプリケーション構築法として、再利用性と拡張性を向上させるためイベントシステムの導入と、それに対応したモジュール化の手法を採用し、連続メディアを対象としたアプリケーションの構築を例にとり、再利用性と拡張性の高いアプリケーションの開発環境を提案する。

この開発環境では、連続メディアを処理しユーザインタフェースを司るモジュールと、モジュール間の制御と通信のために使われるイベントシステムがサポートされる。

また、Real-Time Mach[14] 上で動作する連続メディアアプリケーションツールキット CMT2 を用いて、モジュールを実装し、スクリプト言語としてこれらの開発環境を実装し、新しいモジュール化とイベントシステムの有効性の検証を試みた。

これにより、個々のモジュールは従来のモジュールと違い、メディアデータに互換性があれば様々モジュールと接続可能であり、アプリケーションを生成した後にもモジュールの内容を再構成して、モジュールの機能を拡張できるようになっており、実行中のアプリケーションの機能の拡張も容易に行なえるようになっている。

第 2 章では、連続メディアアプリケーション開発環境の概要と問題点について述べる。第 3 章では、拡張可能なモジュールとい弁システムについて述べ、第 4 章では、モジュールとイベントシステムを用いたスクリプト言語の設計について説明し、第 5 章ではその実装と評価について説明し、第 6 章では、スクリプトプログラミングについて述べる。第 7 章では、考察と将来の課題について述べ、第 8 章で研究全体についてのまとめを行なう。

# 第 2 章

# 連続メディアアプリケーション開発環境

## 2.1 既存の連続メディアアプリケーション開発環境

### 2.1.1 VuSystem

VuSystem[5] は、連続メディアアプリケーションを、連続メディアを処理するコンポーネントである in-band コード部とユーザインタフェースやプログラムのイベント駆動を実行するコンポーネントである out-of-band コード部に分離し、前者には、高速な処理などの高性能性、後者には、プログラミングの容易さといった、コンポーネントの特性に応じたアーキテクチャごとの設計が可能となっている (図 2.1)。

In-band のコンポーネントは、主に連続メディアを処理するために Out-of-band に比較して、システムの資源をより多く使用するとともに、実際の処理には、実時間性や処理時間の制限などが厳しく、処理の効率性やモジュール性に主眼をおいて設計されている。メディアの処理には、フローアーキテクチャが採用され、直接、連続メディアを処理する部分は、複数のモジュールに分割される。これらのモジュールは、生成や接続などの操作に関する共通の規則を持っており、それらを用いてパイプライン的にアレンジされる。

In-band のモジュールは、その入力および出力ポートの有無により、ソース、シンク、フィルタの三種類が存在する。ソースは、出力ポートのみを持ったモジュールであり、通常、連続メディアの入力デバイスのインタフェイスとして使用される。ソースは、入力ポートのみを持ったモジュールであり、出力デバイスのインタフェイスとして使用される。フィルタは、入出力のポートをそれぞれ一つ以上持ったモジュールであり、データの変換や状態の検出といったメディアの処理を行なうのに用いられる。ペイロードとして表現される、これらのメディアのデータは、これらのモジュールのポートを介して、論理的に転送される。

out-of-band のコンポーネントでは、性能よりも開発効率の良さが求められる。In-band で使われるモジュールの作成や制御、およびアプリケーションの GUI に関する制御を行なうため、スクリプト言語 Tcl(Tool Command Language) [8] を用い、Object Tcl[16] のオブジェクトコマンドによりモジュールにコマンドを通達し、in-band からの通知はコー

ルバックの形で out-of-band に返されるようになっている。

VuSystem は、in-band コード部についてモジュール化を行なうことでモジュールプログラミングによる連続メディアアプリケーションの開発の可能性を最初に示し、実装によりその有効性を実証した連続メディアアプリケーション開発環境である。

しかし、in-band モジュールのメソッドに直接関係する out-of-band コード部は分離不可能で、固定されたものとなっている。モジュールをクラスとして扱い、グループ化することはできるが、アクセス手段をあらかじめ講じていない場合は、内部のモジュールへのアクセスは不可能となる。モジュール構築後の再構成はできない。

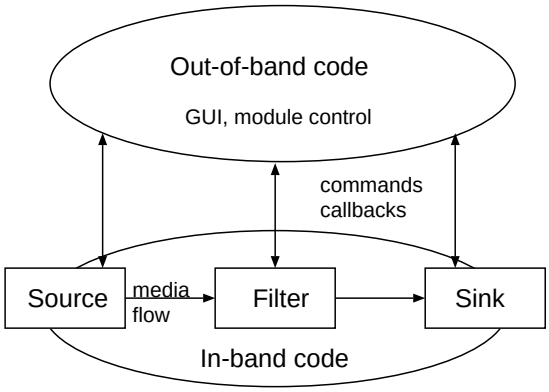



図 2.1: The Structure of Continuous Media Application

### 2.1.2 Medusa

Medusa[3] は、多くのメディアストリームをネットワーク上に擁するような連続メディア環境を提供することのできるシステムである。ネットワーク上に散在する様々なハードウェアのデータを統一して扱い、ハードウェアとソフトウェアの両方をモジュール化して再利用性の向上を計っている。

モジュールは、型を持ったメッセージを送受することで制御やデータの交換を行ない、そのモジュールへの転送は同期的に行なわれる。

信頼性の低いモジュールや保護を行なうためプロキシ機構も備えており、属性の変更をこれでモニタしつつ、保護対象のモジュールへ変更を自分自身に反映する。

また、Medusa では、モジュール生成のタイミングを決定するために様々なモジュールの構成法を検討している。

例えば、メディアストリームの流れに沿ってモジュールを構成する水平切断方式や、メディアストリームのソースやシンクといった垂直方向でまとめ並列化する方式、ディジタル回路のような基礎となる部分を組み上げていく方式、そして、原子・分子モデルによるモジュールの構成などである。これらは階層構造を持つ。また、具体的な階層構造をもつ

くのではなく、モジュールに皮をかぶせることで見かけ上の構造を生成するラッパー方式も、有用な方式として注目している [12]。

現在のシステムでは、水平・垂直カット方式を混ぜたものを採用している [11]。これは、任意の数のソースモジュールやシンクモジュールがある場合でも対応できるようにしたもので、端点の生成を実行後に生成するというものである。

しかし、out-of-band の制御とモジュールの接続の関係により、モジュールの境界を超えたメソッドの呼び出しが行なわれている。

図 2.2 は、従来のマルチメディアアプリケーション開発環境により提供されているモジュールにより構成されるアプリケーション例である。ここでは、Camera モジュールからビデオを表示する xdisplay モジュールへと接続し、カメラの映像をディスプレーに表示するアプリケーションを実現している。

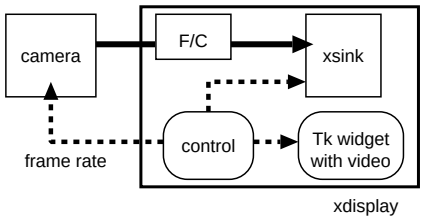



図 2.2: Former Application Structure

ここで、階層モジュール xdisplay は、ビデオストリームデータを受けとり、内部で形式を表示 xdisplay の内部にはさらに形式を変換するフィルタと実際に画面に描画する xsink モジュールがある。

実線で示す in-band に関しては、何らモジュールの接続に問題はないが、破線で示す out-of-band については、xdisplay がコントロールを一括して持っているため、処理速度に応じてフレームレートと設定する要求を camera モジュールに出す場合は、xdisplay モジュールの境界を越えて、直接、camera モジュールのメソッドを呼び出す構成になる。そのため、xdisplay と camera モジュールは、自身の機能に関係なく不可分な存在にされてしまっている。

また、モジュール構築後の動的なモジュールの再構成には対応していない。モジュールの内部で out-of-band コードを持つことはできるが、モジュールとして扱うことはできない。あるモジュール内にある out-of-band コードが外部のモジュールに対してアクセスするための機構、および境界を超えたアクセスを禁止する機構は、備えていない。

## 2.2 アプリケーションの柔軟性を求めて

以上既存の連続メディアアプリケーション開発環境の問題点を、次のようにまとめることがてきる。

- **モジュールの再構成はできない**
  モジュールは、アプリケーションを開発する段階には利用できるが、一旦構築した後は、その内容を変更したり、内部を組み替えることはできない。
- **out-of-band のモジュール化ができない**
  out-of-band は、あくまで in-band モジュールが内部にオプショナルに付随したものという扱いであり、モジュール化ができない。そのため、out-of-band コード部を in-band コード部のようにモジュールの組み合わせて構成することができない。
- **グループ化されたモジュール内部は操作できない**
  グループ化されたモジュールの内部は、基本的にブラックボックスであり、内部の状態を知ることや操作することはできない。そのため、実行中にモジュールに新たな機能を拡張したり、内部のモジュールを置換することでモジュールの機能を変更することも不可能である。

**GUI の複合化**　モジュールとは別にユーザインタフェイスの階層化、複合化についても研究がなされており、これらは、out-of-band のモジュール化として見ることができる [4, 15, 6]。

# 第 3 章

# 拡張可能なモジュールとイベント

## 3.1 拡張可能なモジュール

一般に、アプリケーションの機能のモジュール化は、交換による拡張性を提供するが、独立性の高いモジュールでは、現実のアプリケーションの構築そのものが難しく、独立性が低ければモジュールとしての基本機能である拡張性を欠くというトレードオフが存在する。

連続メディアアプリケーションの構築の場合においても、モジュールプログラミングを採用した多くのツールキット上でこの問題が存在する。メディアストリームの処理に関しては、モジュールごとにきれいに機能を分離することが容易なため、一見、モジュールの独立性の確保には何の問題もないように見える。しかし、現実には、それらのメディアストリームの管理や、モジュールに対する設定等の制御のためにモジュールの境界を超えたアクセスがあり、その結果、特定のモジュール同士の強い依存関係が生じたり、拡張性に制限を与えている。

アプリケーションビルダは、このために、ほとんど機能は同じであるにも関わらず、違うモジュールをわざわざ作成せざるを得ず、再利用性が向上していない。

このように in-band において解決された問題が out-of-band において解決されていないため、モジュールの基本である拡張性が向上していないという問題が存在する。

この解決のために、out-of-band コードのモジュール化を行ない、そのために必要な機構をシステムに採用する。また、モジュールの階層化と内部操作のためのインタフェースの付加することで、従来、グループ化によってブラックボックスとならないようにする。

### 3.1.1 out-of-band のモジュール化

モジュールは、その機能により in-band モジュールと out-of-band モジュールに分類することができる。in-band モジュールは、連続メディアの処理を主目的としたモジュールであり、アプリケーションの核となる。一方、out-of-band モジュールは、in-band モジュールやアプリケーション全体の操作や制御を目的とし、ユーザへのインタフェースを

提供する。従来の方式では in-band のモジュール化のみに着目され、out-of-band については、連続メディアアプリケーションの構築の本質から離れていることもあって、問題にされていなかった。

しかし、in-band と out-of-band は、本来、連続メディアアプリケーションを支える両輪であり in-band のみをモジュール化したところで、拡張性や再利用性の大幅な向上は、望めない。本研究では、in-band 及び out-of-band の両方のコードを等しくモジュールとして扱うことにする。

**in-band と out-of-band を対にする** ある In-band モジュールに特定の関連するウィジットがある場合は、ウィジットの集まりを out-of-band モジュールとして定義し、二つのをまとめて一つのモジュールとして扱うこともできる。例えば、ビデオを読み込むモジュールと、読み込む際のフレームレートを設定するスケールを表示し、設定するモジュールの組合せは、アプリケーションビルダにとっては有用でり、実際に多くの in-band モジュールは特定の out-of-band コード部を持っている。

また、out-of-band コード部は GUI を含めてモジュールとして扱うことで in-band モジュールと同じく拡張性を持ち、階層化することができる。図 3.1では、アプリケーションは、コントロールモジュールとビデオストリームモジュールの二つからなっており、ビデオストリームモジュールはさらに内部に、ソースモジュール等を持っている。

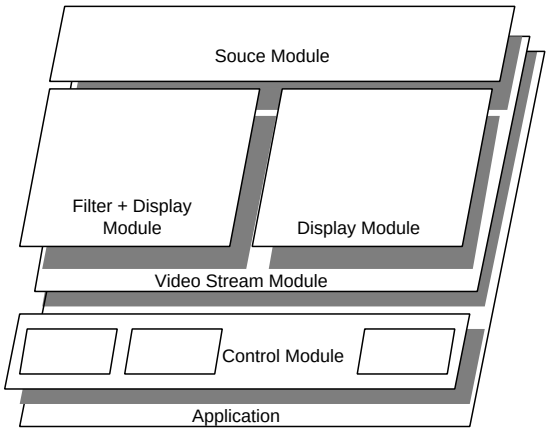



図 3.1: Application and Modules

ユーザ・インタフェースがモジュール化されることにより、これらも in-band モジュール同様に操作の対象となり、例えば、GUI だけでなく、音声で制御するユーザインタフェイスと切替えて使用するようにしたり、様々なユーザインタフェイスを選択的に提供できるようになる。

### 3.1.2 複合モジュール

本研究では、in-band もしくは out-of-band モジュールを二つ以上組み合わせて、より高度な機能を持ったモジュールを複合モジュール (module complex) と呼ぶことにする。

この複合モジュールでは、従来から存在する直列した in-band モジュール群を構築することはもちろん、並列したモジュールの組合せや、階層的なモジュールを作ることも可能である。

また、階層化した場合でも内包するモジュールを操作する手段を提供する。これにより、アプリケーションビルダの必要に応じた機能を提供する。隠蔽去れ、アクセスできなくなったモジュールを持たないことで似たような機能を持った複合モジュールの出現を防ぐ。

再利用性の高い複合モジュールは、複合モジュールライブラリ (MCL) として提供され、アプリケーションの構築や、動的に機能を変更するアプリケーションでその効果を発揮するものである。

## 3.2 イベント

従来の連続メディアアプリケーション開発環境では、モジュールの制御に関しては、モジュールかは考慮されておらず、直接、他のモジュールのメソッドを呼び出す方法がとられていた。

しかし、これではメソッドは、強く依存しているためにアプリケーション実行中にモジュールの拡張や動的な変更はすることができない。

本研究では、直接メソッドを呼び出すのではなく、それらの機能のインタフェースとしてイベントを利用するイベントベースシテスムを採用する。

イベントベースシステムは、複数のコンポーネント間の制御に関する通信をイベントで行なうものである。イベントを発生するコンポーネントは、それを受信して必要な処理を行なうコンポーネントについては一切情報を持っておらず、ただ決められたイベントを発行する。発行されたイベントは、すべてのコンポーネントに対してブロードキャストされる。イベントを受信するコンポーネントでは、自分の興味の対象となるイベントとそれに対する処理をあらかじめ設定して、イベント受信時にその処理を行なう。基本的にイベントを送ってきたコンポーネントがどのようなものかについて関知しない。あるイベントの発信者と受信者の数の関係は、多対多になっている。

また、イベントを使うことで peer-to-peer のモジュールの接続だけではなく、階層化したモジュールの構築も容易になり、従来 In-band と Out-of-band の二つに分離していたアプリケーションのコンポーネントを、自由に切り出して一つのモジュールの中に封じ込んで使用することができるようになる。例えば、画像を処理するモジュールは、その実際の処理だけでなく処理の制御に関する GUI を一つのモジュールの中で持ち、それごと交換するなどの操作ができるようになる。

このシステムでは、各コンポーネントに要求されるのは、イベントを送受信する機能のみであり、コンポーネントが他の特定のコンポーネントのメソッドを呼び出したりするこ

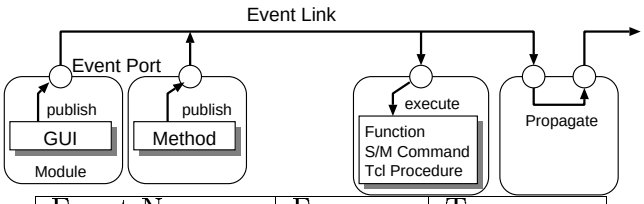
とがないため、高い独立性を維持でき、各コンポーネントは柔軟に運用することができるようになる。

これにより、モジュールの独立性はより高まり、モジュールの置換、挿入や削除といった作業も容易に行なえるようになる。

どのモジュールがどのようなイベントを発行するかは in-band モジュールについては、その機能によりいくつかすでに決まっているものがある。アプリケーション・ビルダもイベントをポート開設することができ、どのモジュールがそのイベントを受けとるかも、記述する。

イベント受信時のアクションには、モジュールメソッドの実行やイベントの再配送を定義できる。

**イベントテーブル**　イベントは、発信する側とそれを受信し適切な処理を行なう側で関係するが、この両者を同時に管理する。イベントテーブルを用いて、どのモジュールがどのイベントに興味があるかを記述する (図 3.2)。




| Event Name | From | To |
|---|---|---|
| start_all | ctrl | cam, xsink |
| stop_all | ctrl | cam, xsink |
| resize_screen | cam, ctrl | xsink |

図 3.2: An Example of the Event Database

### 3.2.1　モジュール内でのイベント

モジュールが外部からイベントを受けとった場合は、そのイベントをモジュール内部のコンポーネントに再配送することができる。また、イベントによりメソッド実行後に、外部にイベントを発行する場合は、あらかじめ定義した出力イベントポートにイベントを発行するように定義する。

### 3.2.2　モジュールの独立性の確保

図 2.2 で示したような、モジュールの境界を超えた不正なメソッド呼び出しは、イベントシステムの導入により、図 3.3 のように解決することができる。

このアプリケーションは、start ボタンを押すとカメラが映像を送りだし、それをディスプレイに表示するものである。まず、メディアデータ全体の流れを制御する start イベントを持つ control モジュールは独立した存在になっている。そして、各モジュールは control からのイベントを受けるように入力イベントポートを持っている。

アプリケーションは、大きく分けて三種類のパーツから構成される。一つはモジュールで、このアプリケーションでは動画を入力する camera、ユーザからのコマンドを受け付けそれを他のモジュールに伝える control、動画の形式を表示可能なものに変換し表示する xdisplay というモジュールから成っている。二つめはメディアで、図中で実線となっているものである。モジュールのポート間を接続し、その方向に連続メディアを流して処理することになる。最後は、イベントでモジュール間で制御を行なうものである。このアプリケーションでは、control から start_all と stop_all のイベントが使用されており、camera と xdisplay( この中では xsink) がこのイベントに注目していることを表している。モジュールは、イベントを受けとった場合、さらに別のイベントを発生したり、あるいはモジュール内部のメソッドを実行するなどを指定する。イベントの発生はイベントコマンドにより実現される。

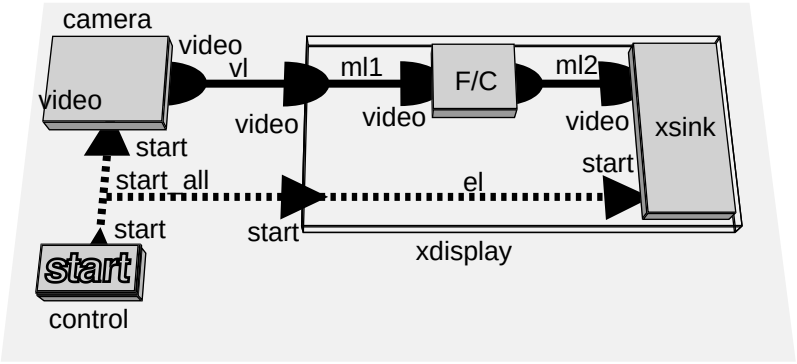



図 3.3: Our Application Structure

このようにモジュール xdisplay とカメラのモジュールの間は、イベントの送受によって成り立っており、例えば、カメラ・モジュールが別なモジュールへと置換されり、あるいは、モジュール xdisplay 内のコントロールが変更されても、互いに相手の変更を必要としない独立した関係を保っている。

## 3.3 イベント・モジュール・プログラミング

**プログラミングにあたって** ユーザはモジュールによるプログラミングを行なうに当たっていくつかのことを理解していなくてはならない。一つは、モジュールのストリームリンクをどのように接続するかということである。もう一つは、モジュール間のイベントの伝達をどのようにするか、である。

**実行中のモジュール操作** モジュールは、実行中に交換することができる。これを利用することで、開発直後の信頼性の低いモジュールをそのまま、実用アプリケーションに使うことができる。モジュールを開発した場合にエラー発生時にモジュールを交換することを

あらかじめ設定しておく使い方も考えられる。モジュール内で致命的で外部に悪影響を及ぼすと判断したモジュールを自動的に、以前のモジュールに戻してしまうのである。

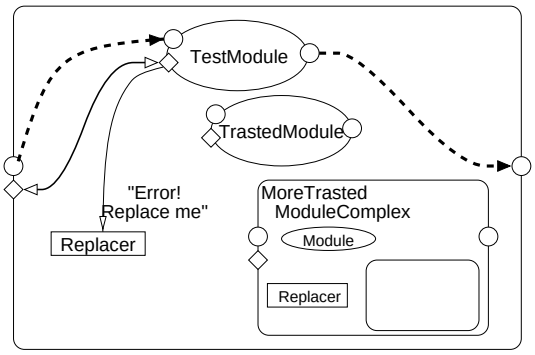



図 3.4: An Error Occured at Test Module

図 3.4、3.5 では、信頼性の高い複合モジュールを示している。TestModule が開発されたばかりでバグを含んでいる可能性の高いモジュールである。TreastedModule は、機能は劣るものの、稼働実績のある信頼性の高いモジュールであり、MoreTrasted Module Comptex は、さらにバージョンが古い同じく稼働実績のあるモジュールを含んでいる複合モジュールである。

TestModule が、自分で解決し得ない、アプリケーション全体に影響を及ぼす致命的なエラーが発生すると、モジュール置換部にエラーをイベントで通知する (図 3.4)。

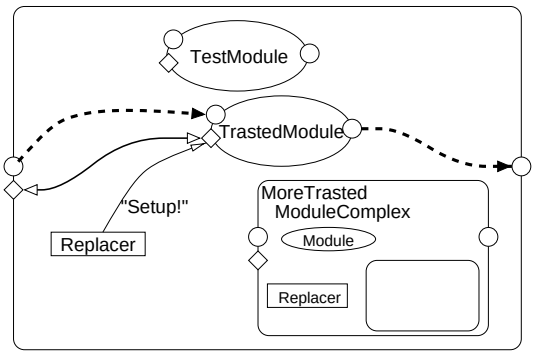



図 3.5: Replace Errored Module

モジュール置換部は、次にストリームとイベントのリンクを接続すべきモジュールを決定し[1]、セットアップを行ない、リンクを切替える[2](図 3.5)。

[1] ここではじめて該当するモジュールを生成しても構わない
[2] このあとで古いモジュールを消去しても構わない

**モジュール内コンポーネントの操作**　モジュール、リンクやイベントはすべて生成後の操作により、変更することができる。例えば、あるボタンを押すことで不要なモジュールを削除したり、新たにモジュールを挿入することはもちろん、大幅なモジュールの操作によりアプリケーションの本質的な機能も変更可能となる。

# 第 4 章

# スクリプト言語 Rtm の設計

## 4.1 概要

第 3章で提案したモジュールとイベントシステムを用いたアプリケーション開発環境の有効性を実証するために、これらに基づいた開発環境としてスクリプト言語 Rtm の設計および実装を行なう。本章では、このスクリプト言語の設定について述べる。

スクリプト言語 Rtm は、以下の概念を提供する。

**モジュール** メディアストリームを処理し、GUI を持つなどアプリケーションの根幹をなす。ユーザは、このモジュールに適切な指示を与え、これを連結することでアプリケーションを構築する。

**メディア** モジュールの持つメディアポートをメディアリンクで接続することによりモジュール間をメディアストリームが流れる。

**イベント** モジュールが持つイベントポートから発生し、イベントリンクにより接続された複数のイベントポートに通知する。イベントを受けとったモジュールは、内部のメソッドを実行したり再配送する。

**スコープ** モジュールを単位としてスコープが存在し、変数やイベントなどはこの範囲を超えることはできない。

## 4.2 モジュール

モジュールは、内部にモジュールの基本機能を収めたメソッドまたはコードを持つ。また、その制御のために制御の種類ごとのイベントポートを持っており、このイベントポートにイベントを与えることで、それらの機能を実行することができる。与えるイベントには、引数を持たせることもできる。

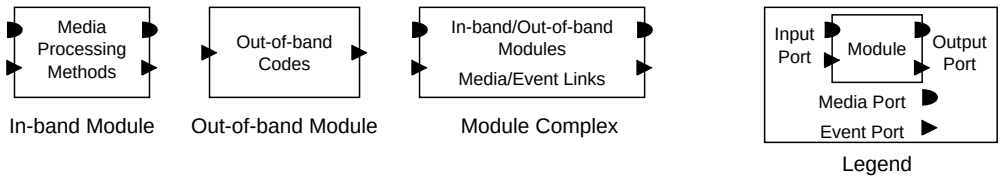



図 4.1: Modules on Toolkit Rtm

連続メディアを処理するモジュールは、これ以外にメディアを処理するメディアポートを持つ。メディアポートは、ビデオやオーディオといったメディアのタイプを持ち、他のモジュールからのメディアストリームの入力、または主力を行なうインタフェースとなる。

モジュールは、複合モジュールとして内部に階層的に複数のモジュールを持つことができる。

モジュールのデータ構造は、次のようになっている。

- **モジュール ID**：モジュールに与えられる固有の整数値
- **パス名**：モジュール階層内の位置を含むモジュール名。例えば、モジュール A に属するモジュール B のパス名は、`.A.B` と表現する。
- **モジュール固有情報**：そのモジュールタイプに固有のデータ構造。モジュールのタイプに依存して決まるデータ構造への参照。モジュールが別のモジュールを内包する複合モジュールの場合には、内包モジュールを参照するモジュール固有情報を持つことになる。
- **直属のストリーム ID**：モジュールが直接に属するストリームの ID。

現在ところ、サポートしているモジュールは、機能別に三種のモジュールが存在する。連続メディアを処理することを目的とした in-band モジュールと、モジュールもしくはアプリケーションの制御を目的とした out-of-band モジュール、そしてそれらを組み合わせたモジュールである。以下にそれらを説明する。

**in-band モジュール**　連続メディアの処理だけを目的にしたプリミティブなモジュールである。モジュールがデフォルトで持つ、内部のメソッドとその制御のためのイベントポートの他に、メディアのタイプのついた入出力のためのメディアポートを持っている。

具体例として、VideoCrasSrc モジュール (A.1節 p. 41) を挙げる。このモジュールは、CRAS [13] を介して、QuickTime 形式 [1] のムービーファイルを読み込むモジュールである。

その制御のためのイベントポートには、ムービーファイル名を指定するもの、バッファサイズを指定するもの、ムービーのタイプを出力するもの、そしてムービーの再生終了を通知するものがあり、ムービーのメディアストリームをビデオタイプとして出力メディアポートを持っている。

**out-of-band モジュール**　モジュールの操作や制御を目的としたもので、ユーザインタフェースを含むこともある。この場合はウィジットを用いてユーザからの入出力の制御を受け付けたり、表示したりする。モジュールの外部とのコネクションはイベントにより行なわれる。

**その他のモジュール**　その他にも、アプリケーションを制御するポート持たないモジュールや、キーボードやマウスなどに変わる入出力デバイスとしてのモジュールも考えられる。

**複合モジュール**　一つ以上のモジュールを内部に含むモジュールである[1]。例えば、in-band モジュールとその制御関係する out-of-band モジュールを組み合わせて、一つのモジュールとして扱うこともできる。このような組み合わせ方は、連続メディアの処理とそれに付随するユーザ・インタフェースを一つの単位として、置換や削除するなどの操作ができる。

複合モジュールは、イベントリンク、メディアリンクで接続された複数のモジュールを内包し、入出力ポートを持つことで、一つのモジュールとして機能の抽象化して扱うことができる。

複合モジュールは、内包するモジュールの生成とそのイベントリンク及びメディアリンクの接続をデータ構造を次に示す。

- **内包するモジュールのテーブル**
- **内包するイベントリンクのテーブル**
- **内包するメディアリンクのテーブル**

**ストリーム**　複合モジュールのうち、メディアリンクがソースモジュールからシンクモジュールまでつながったものは、ストリームとして特別に扱うことができる。ストリームとなったモジュールに対しては、アプリケーションとして実行するためのコマンドを発行することができるようになる。

最終的なアプリケーションは、一つ以上のストリームから構成されることになる。

## 4.3　メディア

モジュールのメディアポート間を接続するリンクをメディアリンクと呼ぶ。

メディアストリームが流れる環境を準備するには、図 4.2 に示す通り、メディアリンクを生成し、次いでその両端をモジュールのメディアポートに接続する。リンクの接続先のポートを変えることでメディアストリームの流れを変えることもできる。メディアリンクの接続は、入力、出力ともに一つである。

[1] 複合モジュール (Module Complex) は、0 個以上の In-band モジュール、Out-of-band モジュールか Out-of-band コード部を含んだもので外部からは一つのモジュールとして見えるものである (まったく何もない複合モジュールは NULL モジュール複合体である)。メガモジュールとか、グルーピングされたモジュールとも言う。最終的にはアプリケーションも一つの複合モジュールとしてみることができる。特に、In-band と Out-of-band の二つの関連するモジュールもしくはコード部をそれぞれ一つ以上もち、両者が同時に機能するモジュール複合体を混成モジュール (Combined Module) と呼ぶ。

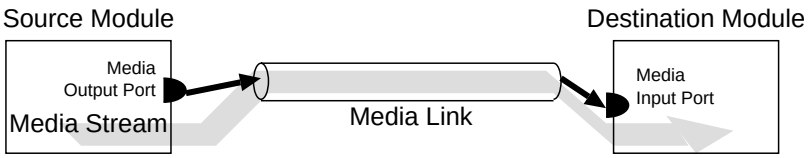



図 4.2: Media Link

## 4.4 イベント

イベントは、アプリケーションやモジュールの制御を司る。メディアと同様にモジュールの持つイベントポートをイベントリンクで接続することによりに、利用できる環境が準備される。入出力の数の関係は、多対多である。モジュール内部のメソッドがイベントポートに対して、イベントの発行を指示すると、イベントリンクで接続された出力側のポートに一斉にイベントの発行が通知される。イベントに引数をつけることで、データを他のモジュールに伝送できる。

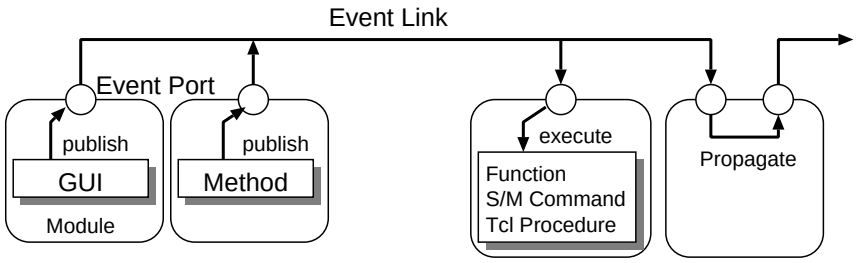



図 4.3: Event

## 4.5 まとめ

モジュール、イベントそしてメディアを用いたアプリケーションの構築例を図 4.4 示す。

このアプリケーションでは、大きく二つのモジュールに分かれる。一つは、複数のモジュールを内包している複合モジュールでメディアストリームのソースからシンクまでを含んでいるため、ストリームとして扱われている。もう一つは、Out-of-band モジュールで、ストリームに対する制御を行なう部分である。

破線は、イベントの接続を示している。この例では、ストリーム内のシンクモジュールに対してイベントがつながっており、ストリームの内部のソースモジュールは、Out-of-band と in-band モジュールが対になっているモジュールである。例えば、GUI のボタンウィジットを押すことで「再生」や「停止」といった指示をこの out-of-band モジュールから

対となっている in-band モジュールに伝達し、ソースモジュールからは、同様の命令がシンクモジュールに配送される。

実戦で示すメディアの流れは、ソースモジュールからシンクモジュールに一直線であり、その間にフィルタモジュールが入っている。フィルタは、イベントの入出力のない構造的にシンプルなモジュールとなっている。

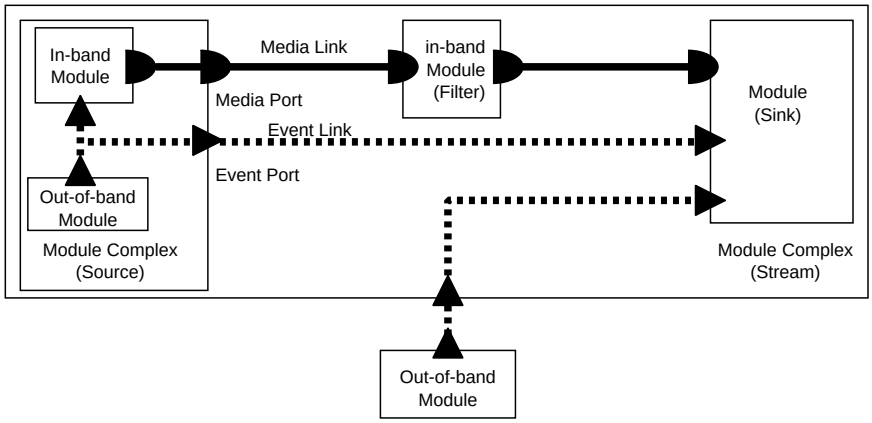



図 4.4: Rtm Application Development Environment

# 第 5 章

# スクリプト言語の実装

## 5.1 概要

スクリプト言語 Rtm は、スクリプトを記述するだけで連続メディアアプリケーションを構築できることを目的とした開発環境である。これは大きく二つの部分から構成されており、一つはアプリケーション・ビルダが作成したスクリプトを解釈し、実行するスクリプトインタプリタであり、もう一つは、連続メディアを処理するための in-band モジュール群となっている。

スクリプト言語の開発に当たっては、シェルスクリプト言語としての機能と、GUI の利用のために Tcl/Tk を利用し、in-band モジュールの開発には、連続メディアツールキット CMT2 を利用している。

**連続メディアツールキット CMT2**　CMT2 は、Real-Time Mach 上で非常に繁雑な連続メディアを処理するための機構と、実際に連続メディアを処理するためのモジュールを持っており、連続メディアアプリケーション構築に必要な機能を備えている。

モジュールとイベントの提供により、アプリケーションのコンポーネントの独立性を高め、容易に高い柔軟性を持つアプリケーションを開発できる開発環境を構築できることになる。

なお、本研究の開発環境は、実際の連続メディア処理を行なうための機能を提供するツールキット CMT2 の上に構築されており、図 5.1 に示すような構成となっている。

**Tcl/Tk**　シェルスクリプト言語としての基本機能を持ち、言語のコマンドの拡張のためのインタフェースを提供しているという特徴を持っており、この上で、マルチメディア処理のためのシステムを構築することで、効率良く、スクリプト言語を開発することが可能であるため、これを利用した。このツールキットはスクリプト言語 Tcl/Tk を拡張したものであり、連続メディアの処理を目的したモジュールを操作し、アプリケーションを構築するために、`module`、`media` そして `event` の三つのコマンドを用意している。

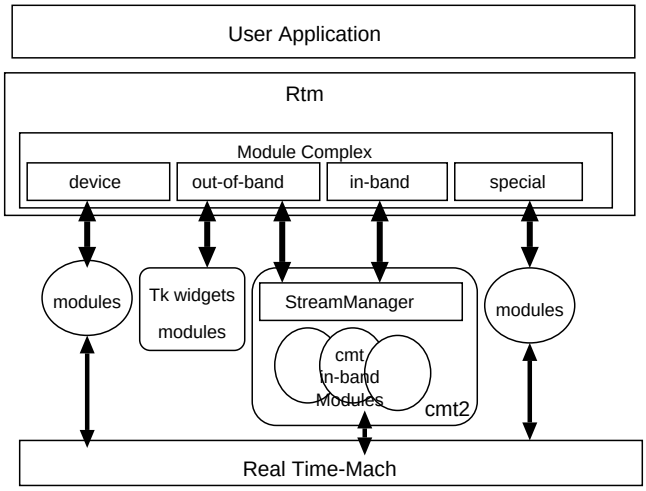



図 5.1: Rtm and Cmt

表 5.1: Main Data Structure of Rtm(`RtmInfo`)

| Type | Valuable | description |
|---|---|---|
| `Tcl_HashTable *` | `moduleTable` | |
| `Tcl_HashTable *` | `elinkTable` | |
| `Tcl_HashTable *` | `eportTable` | |
| `Tcl_HashTable *` | `mlinkTable` | |
| `Tcl_HashTable *` | `mportTable` | |
| `char *` | `cwPath` | カレントワーキングパス |
| `StreamManager *` | `sm` | ストリームマネージャ |
| `Tk_Window` | `mainWin` | ウィンドウ情報 |

**スコープの実装**　イベント、メディア、モジュールはすべてスコープを持つ。内部で実行されたことは、そのスコープないでしか有効ではない。実装法は、いくつかあって、一つは、オブジェクトコマンドを内部でスイッチする方法、もう一つは、オブジェクトコマンドを使用せずにカレントワーキングパスを見て、該当するオブジェクトを選びだし、適切な処理をする方法である。後者の方が早く実装できるので、こちらを選択する。

実装上の問題点として、イベント、メディア、モジュールなどの拡張部分にしか、スコープが適応されない。例えば、内部で、計算のための変数を何か用いた場合は、ツールキットのスコープには関係ないスコープを持つ変数になってしまう。

## 5.2　モジュールコマンド

すべてのモジュールは、表 5.2 に示すデータ構造 `RtmModule` を持つ。

表 5.2: Data Structure of Module(`RtmModule`)

| Type | Valuable | description |
|---|---|---|
| `ModuleID` | `id` | モジュール ID |
| `char *` | `name` | パス名 (モジュール名) |
| `RtmModuleTypeInfo *` | `info` | モジュール固有情報 |
| `RtmInfo *` | `rtmPtr` | 所属するインタプリタのデータ |
| `Tcl_Interp *` | `interp` | 所属するインタプリタ |
| `int` | `smId` | 直属のストリームの ID |

**入出力**　モジュールは、メディアストリームの入出力インタフェースとして、メディアポートとモジュールを制御し、あるいはその状態を通知するためのイベントポートを持つ。

モジュールのポートはモジュール生成の際に暗黙のうちに生成され、そのポートにバインドされた機能によりデフォルトのポート名が与えられる。例えば、ビデオポートなどは複数ある場合は video0、ない場合は、video などとなる。

### モジュールの種類

**in-band モジュール** in-band モジュールは、CMT2 が低起用する連続メディアツールキットにより作成したモジュールである。in-band モジュールには、モジュール自身を制御するための out-of-band コードや out-of-band モジュールは含まれておらず、最低でも一つのストリームポートを持っている。また、モジュールの制御のためのインタフェースとしてイベントポートが提供されている。このイベントポートに適当なイベントを送ることにより内部のアトリビュートの設定などができるようになっている。

in-band モジュールの生成や操作やリンクについては、ストリームマネージャが管理するため、ストリームマネージャの提供するインタフェイスを使用して、実装する。イベントベースシステムについては、ストリームマネージャでは扱わないため、イベント管理表をコマンドとコールバックという形式に変換してストリームマネージャに渡す。

CMT2 では、CMT2 のモジュールは、連続メディアの処理に関して clock_port や reserve_port などの mach_port をアトリビュートの設定を要求するが、Rtm ではこれを内部で設定、処理することで、アプリケーションビルダにより設定は不要なものとした。

```
module .cam -type camera {
  media port .video -output
  media port .video bind Video
  event port .fps -output
  event link .fps bind fps
}
module .display -type Xdisplay
```

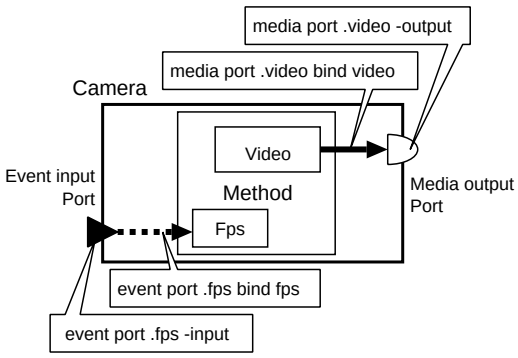



図 5.2: Example of Module Scripting

**モジュールコマンドの詳細** スクリプト言語で利用可能な module コマンドは表 5.4のようになっている。

例を図 5.2に示す。最初の四行では、camera モジュールを . cam というインスタンス名で生成しており、初期設定として、イベント.start_all と.stop_all を受けとった場合に、それぞれモジュール内のメソッド start と stop を実行するように設定している。五行目では、カメラの撮影レートを毎秒 10 枚に設定している。同様に、モジュール xdisplay とモジュール control の生成を行なっている。

**複合モジュール** モジュールの内部にモジュールやスクリプトが存在する階層構造を持つものを生成することができる。GUI などの out-of-band に関係するスクリプトのみを持つものは、out-of-band モジュールとなる。 また、単なるグループ化ではなく、異なる性質のモジュールを組合せ、大きな機能とする複合化が機能であり、それらは、複合モジュールとして扱われる。

複合モジュールの固有情報のデータ構造を表 5.3 示す。

**out-of-band モジュール** out-of-band モジュールは、複合モジュールと同じ手法により作成される。

表 5.3: Data Structure of Module Complex

| Type | Valuable | Description |
|---|---|---|
| Tcl_HashTable * | moduleTable | 内包するモジュールのテーブル |
| Tcl_HashTable * | elinkTable | 内包するイベントリンクのテーブル |
| Tcl_HashTable * | mlinkTable | 内包するメディアリンクのテーブル |
| Tk_Window | tkwin | Tk ウィンドウ構造体 |
| Display * | display | X ディスプレイ構造体 |
| int | updatePending | 更新 |

イベントは、そのイベントポートを所有するモジュールからのみアクセスできるスコープを持っている。モジュール内で定義されたイベントポート名は、他のモジュールの同名のイベントポート名とは区別される。

図 5.3に示す例では、 イベント .start が定義されているが、これは、モジュールタイプ GUI のある生成されたモジュールの中でのみ有効なアクセス可能なイベントである。

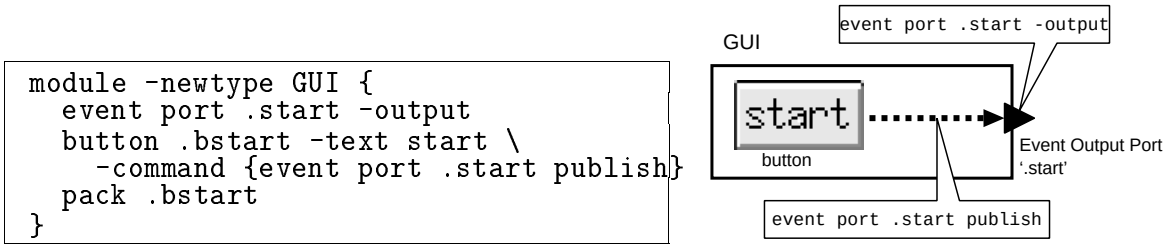



図 5.3: Example of Out-of-band Module Scripting

モジュールが内部に持つスクリプトは、モジュールの生成時に評価され、イベントポートなどは、テーブルに入れられ、表 5.3に占め去られる構造体に代入される。

**複合モジュール** 例を示す。モジュール xdisplay は、次のように動画の形式を変換するフィルタとそれをディスプレイに出力するシンクの二つからなるモジュールとしてあらかじめ定義されている。

**ストリーム** ストリームは、特別な複合モジュールであり、ここでは、特殊なイベントの生成が可能となる。例えば、ストリームに対しては、

**コントロール** 特殊なモジュールの二つ目として、コントロールモジュールがある。これは、Cmt2 におけるコマンドを設定するイベントを持つモジュールである。

```
module -newtype xdisplay {
  module .fc -type formatConv
  module .xsink -type xsink {
    .start_all bind -method start
    .stop_all bind -method stop
  }
  media link .l1 -from .in.video \
    -to .fc.video
  media link .l2 -from .fc.video \
    -to .xsink.video
  pack .xsink
}
```

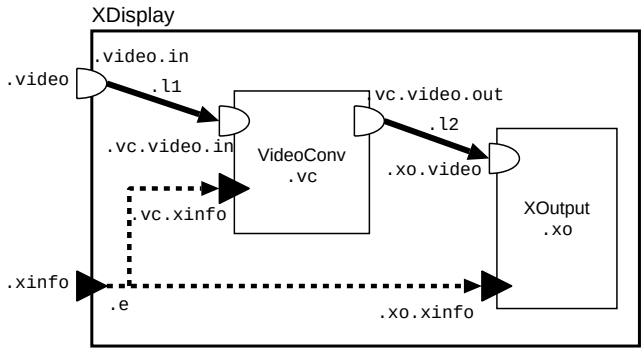



図 5.4: Example of Module Scripting

```
module -newtype Control {
  module .gui -type GUI
  event port .start -control Start \
    -delivery order
  event port .stop -control Stop \
    -delivery order
  event port .offset -control Offset \
    -delivery order
  pack .gui
}
module .ctrl -type Control
module .s1 -type Stream1
module .s2 -type Stream2
event link .start -from .ctrl.start \
  -to .s1.start -to .s2.start
event link .stop -from .ctrl.stop \
  -to .s1.stop -to .s2.stop
event link .offset -from .ctrl.offset \
  -to .s2.offset
```

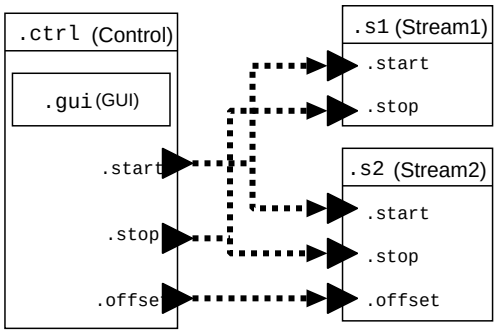



図 5.5: Example of Module Scripting

コントロール系のイベントをどこが実行するかという実装上の問題がある。通常、イベントの処理は、イベントを受けとったモジュールで行なうが、コントロールに関しては、コントロールモジュールが処理し、CMT2 の関数 `CmtCommandCall` を実行する。これは、関数が直接、ストリームを引数にとって実行できるため、こちらでわざわざモジュールに配布して、それぞれストリームごとに実行するという手続きを踏まないようにしたためである。

ストリーム系イベントポートは内部に実行すべきメソッドのないダミーポートとなり、イベントのリンクについても、コントロールからしか接続できないという仕様上の制限が発生する。

ただし、ストリーム系イベントの到着を待って処理をすることを可能にするために、イベントは配送される。

**Rtm コマンド** 'module' スクリプト言語 Rtm 上で利用できる `module` コマンドを図 5.4 にまとめる。

表 5.4: Module Command

| |
|---|
| `module` *pathName* `-type` *moduleType*<br>モジュール型 *moduleType* のモジュールを、*name* として生成する。モジュール型は、インタプリタにあらかじめ組み込まれたもの以外に、ユーザが定義したものも記述できる。 |
| `module -newtype` *typeName* { *module complex definition* }<br>モジュールの定義部にしたがって生成、構築されたモジュールを内包する新しいモジュールタイプを定義する。 |
| `module` *pathName* `-modify` { *module complex modification* }<br>すでに生成されたモジュールの内容を変更する。 |
| `module types`<br>定義されているモジュールの一覧 |
| `module` *pathName* `destroy`<br>モジュール削除 |
| `module` *pathName* `configure` *?option..?*<br>モジュールのアトリビュートの設定 |
| `module` *pathname* `info`<br>モジュール内部情報の取得 |

## 5.3 メディアコマンド

メディアコマンドは、連続メディアを処理するモジュールを接続する。

**メディアポート**　メディアポートは、表 5.5に示すデータ構造 RtmMport を持つ。

表 5.5: Data Struccure of Media Port(RtmMport)

| Type | Valuable | description |
|---|---|---|
| LinkID | id | モジュールの内部でユニーク CMT 共用 |
| char * | name | ポートのパス名 |
| int | direction | RTM_OUTPOUT(入力) か RTM_INPUT(出力) |
| RtmModule * | modulePtr | ポートを所有モジュールへのポインタ |
| RtmInfo * | rtmPtr | メイン情報へのポインタ |

メディアポートに関するメソッドを、表 5.6 に示す。

表 5.6: Methods of Media Port

| |
|---|
| Rtm_MportCreate(Tcl_Interp *interp, RtmInfo *rtmPtr, char *mportName, int direction)<br>mportName を持ったメディアポートを生成する。 |
| Rtm_MportDestroy(RtmInfo *rtmPtr, char *name)<br>イベントポートを削除する |

メディアポート名は、メディアのタイプに対応して付けられる。例えば、ビデオメディアストリームを扱うメディアポートは、video というポート名が与えられる。デフォルトでは、ポート名は、video0 であり、0 は省略可能となっている。また、入出力に関しては、ポート名の最後に In または、Out を加えることで、表現する。

**メディアリンク**　データ構造を表 5.7に示す。

表 5.7: Data Struccure of Media Link(RtmMlink

| Type | Valuable | description |
|---|---|---|
| LinkID | id | ID for CMT |
| char * | name | パス名 |
| RtmMport | input | 受信ポートテーブル (キーは name) |
| RtmMport | output | 送信ポートテーブル (キーは name) |
| Tcl_HashTable * | elinkTable | イベントリンクテーブル |

メディアリンクに関するメソッドを、表 5.8 に示す。

表 5.8: Methods of Media Link

| |
|---|
| `Rtm_MlinkCreate(RtmInfo *rtmPtr, char *name)`<br>`name` を持ったメディアリンクを生成する。 |
| `Rtm_MlinkDestroy(RtmInfo *rtmPtr, char *name)`<br>`name` を持ったメディアリンクを削除する。 |
| `Rtm_MlinkConnect(RtmInfo *rtmPtr, char *elinkName, char *eportName, int direction)`<br>`elinkName` を持ったメディアリンクに `direction` の向きを持ったメディアポート `eportName` を接続する。`direction` は、入力元 (`RTM_INPUT`) か、出力先 (`RTM_OUTPUT`)。 |
| `Rtm_MlinkDisconnect(RtmInfo *rtmPtr, char *elinkName, char *eportName)`<br>`elinkName` を持ったメディアリンクからメディアポート `eportName` へのリンクを切断する。 |

**Rtm コマンド** 'media' スクリプト言語 Rtm で利用可能な `media` コマンドは表 5.9のようになっている。

**例** Rtm スクリプト記述例として、メディアリンク .link1 は、モジュール .cam の video ポートと. xsink モジュールの video ポートを接続している。

```
media link .link1
.link1 connect .cam.video to .xdisp.video
```

表 5.9: Media Command

| |
|---|
| `media link` *pathName*<br>リンクをリンク名 *name* として生成する。 |
| `media link` *pathName* `destroy`<br>リンクを削除する。 |
| `media link` *pathName* *?*`-from` *.module0.port?* *?*`-to` *.module1.port?*<br>リンクを接続する。リンク元のポート*.module0.port* およびリンク先のポート*. module1.port* を指定する。どちらか片方だけの接続もできる。また、すでに接続されているリンクとは異なるリンク先もしくはリンク元ポートを指定した場合は、繋ぎ替えが行なわれる。 |
| `media link` *pathName* `disconnect` *?*`-from` *.module0.port?* *?*`-to` *.module1.port?*<br>リンクを切断する。`media connect` と同様に両方もしくはどちらか片方のみの切断ができる。 |
| `media link` *pathName* `links`<br>リンクの内包するオブジェクトを取得する。 |
| `media link` *pathName* `sync` *name*<br>同期を取る |

## 5.4 イベントコマンド

イベントに関するコマンドは、イベントポートに対するものとイベントリンクに対するものと二種類がある。それぞれオブジェクトの生成と削除を持ち、イベントポートは、イベントを発行するメソッド、そして、イベントリンクは、リンクの接続と切断に関するメソッドを持つ。

**イベントポート** モジュールは、イベントポートを介してイベントの受け付けや発行を行なう。外部からイベントを受け付けるとモジュール内ではあらかじめ定義されているメソッドを実行するか、あるいは内部に向けてさらにイベントを発行することができる。

**イベントの発行** 内部のメソッドやユーザインタフェース[1]からイベントを発行することができる。ユーザインタフェースは、イベント発行命令[2]を使用する。

**イベントの受信** イベントポートには、デフォルトで定義されているものが二つ存在する。in-band モジュールが start ポートと stop ポートでイベントを受けとった場合、直に start メソッドと stop メソッドを実行する。

また、これ以外にもイベントとモジュール内部のメソッドの実行を結ぶことができる。それ以外にも、グループ化されたモジュールではイベントをさらに内部のモジュールに伝搬するために、新たにイベントを発行することができる。

[1] スクリプト言語上ではウィジット
[2] スクリプト言語上では、event コマンドの publish アクション

**イベントポート**　イベントポートのデータ構造を表 5.10 に示す。

表 5.10: Data Structure of Event Port(RtmEport)

| Type | Valuable | description |
|---|---|---|
| int | id | ID |
| char * | name | パス名 |
| int | type | ポートの型 |
| RtmModule * | modulePtr | ポートを所有しているモジュール |
| int | (*to)() | バインドしている関数 |
| Tcl_HashTable * | eportTable | イベントポートテーブル |

イベントポートに関するメソッドを、表 5.11 に示す。

表 5.11: Methods of Event Port

| |
|---|
| Rtm_EportCreate(Tcl_Interp *interp, RtmInfo *rtmPtr,char *eportName, int type)<br>mportName を持ったイベントポートを生成する。 |
| Rtm_EportCtrlCreate(Tcl_Interp *interp, RtmInfo *rtmPtr,int argc, char *argv)<br>mportName を持ったコントロール系イベントポートを生成する。 |
| Rtm_EportDestroy(RtmInfo *rtmPtr, char *name)<br>イベントポートを削除する |
| Rtm_EportPublish(RtmInfo *rtmPtr, char *name, int argc, char *argv[])<br>イベントポートにイベントを発行する |

**イベントリンク**　イベントリンクのデータ構造を表 5.12に示す。

表 5.12: Data Structure of Event Link(RtmElink

| Type | Valuable | description |
|---|---|---|
| int | id | ID |
| char * | name | パス名 |
| Tcl_HashTable * | senderTable | 送信者ポートテーブル |
| Tcl_HashTable * | recipientTable | 受信者ポートテーブル |
| Tcl_HashTable * | elinkTable | |

イベントリンクに関するメソッドを、表 5.13 に示す。
スクリプト言語 Rtm で利用可能な evnet コマンドは表 5.14のようになっている。

表 5.13: Methods of Event Link

| |
|---|
| `Rtm_ElinkCreate(RtmInfo *rtmPtr, char *name)`<br>`name` を持ったイベントリンクを生成する。 |
| `Rtm_ElinkDestroy(RtmInfo *rtmPtr, char *name)`<br>`name` を持ったイベントリンクを削除する。 |
| `Rtm_ElinkConnect(RtmInfo *rtmPtr, char *elinkName, char *eportName, int direction)`<br>`elinkName` を持ったイベントリンクに `direction` の向きを持ったイベントポート `eportName` を接続する。`direction` は、入力元 (`RTM_INPUT`) か、出力先 (`RTM_OUTPUT`)。 |
| `Rtm_ElinkDisconnect(RtmInfo *rtmPtr, char *elinkName, char *eportName)`<br>`elinkName` を持ったイベントリンクからイベントポート `eportName` へのリンクを切断する。 |

表 5.14: Event Command

| |
|---|
| `event port` *pathName* *?*`-control` *controlName* `-delivery` *type ?*<br>イベントをイベント名 *pathName* として生成する。`-control` オプションが指定された場合は、ポートはコントロールコマンドとして機能し、そのイベントの配送法は、*type* となる。 |
| `event port` *pathName* `destroy`<br>イベント *pathName* を削除する。 |
| `event port` *pathName* `publish` *? argument ? ...?*<br>イベントをイベントポート *pathName* に対して発行する。 |
| `event link` *pathName*<br>イベントをイベント名 *pathName* として生成する。 |
| `event link` *pathName* `destroy`<br>イベント *pathName* を削除する。 |
| `event link` *pathName* `connect` *?*`-from` *.module0.port? ?*`-to` *.module1.port?*<br>イベントを接続する。リンク元のイベントポート *.module0.port* とリンク先のイベントポート *.module1.port* を指定する。どちらか片方だけの接続もできる。また、すでに接続されているリンクとは異なるリンク先もしくはリンク元ポートを指定した場合は、繋ぎ替えが行なわれる。 |
| `event link` *pathName* `disconnect` *name ?*`-from` *.module0.port? ?*`-to` *.module1.port?*<br>リンクを切断する。`event connect` と同様に両方もしくはどちらか片方のみの切断ができる。 |

**イベントの引数**　イベントポートに対して発行されるイベントは引数を持つことができる。この引数は、Tcl スクリプトとして記述することで明示的に渡されるものと、実際にあるが、スクリプト上では表記しない引数との二つがある。明示できに渡す時にはイベントの発行の際にこれを publish アクションの後に続けて記述する。

**デフォルトポート名**　イベントを受けとるとそれだけでモジュール内のメソッドを実行するデフォルトのポート名がモジュール生成時に設定される。それらを変更することも可能となっている。destory ポートはモジュールを消滅させる。

**コントロール系イベント**　ストリームを操作するための特殊なイベントとして、コントロール系イベントが存在する。これは、CMT2 においてストリームマネージャにコマンドをしているもので、モジュール Control の中でのみ指定できるイベントポートである。イベントポートの生成は、関数 CmtCommandCreate の実行につながっている。コントロール系イベントの実行には、配送方式を-delivery オプションで指定できる。

受信側のイベントポートは、モジュール内部に実行するメソッドを持たないことになり、RTM_CONTROL という識別子が与えられる。

このイベントのリンクは、コントロールモジュールからストリームに対してしか接続することができない。ただし、接続先のイベントポートには、イベントは配送される。

次のように記述する。

```
event port .start -control Start -delivery order
```

**例**　図 5.6の例では、GUI モジュールの start ボタンを押すと、CRAS モジュールの movie_start コマンドが実行されるように、イベントを設定している。

```
module -newtype GUI {
  event port .start
  button .b1 -text start -command {event port .start publish}
  pack .b1
}

module .gui -type GUI
Module .src -type VideoCrasSrc
event link .start connect -from .gui.start -to .src.start
```

図 5.6: Event Scription

# 第 6 章

# スクリプト・プログラミング

## 6.1 GUI の記述

モジュール control は、GUI だけを持つモジュールとなっている。このモジュールではボタンを押すことでそれぞれ.start_all と stop_all の二つのイベントが発生する。イベントが発生した時にどのモジュールが何を実行するのかは、そのイベントに注目しているモジュールの側で設定する。

同じイベントを発生することだけ留意して、新たな GUI モジュールを定義しておけば、複数の GUI を切替えて使用することもできる。また、実際のメディアの処理を行なうモジュールと GUI とを一つのモジュールとすることで、そのモジュール変更時に機能とともに GUI も変更されることになる。

```
module -newtype control {
  global .start_all .stop_all
  button .b1 -text start -command {.start_all occur}
  button .b2 -text stop  -command {.stop_all occur}
  pack .b1 .b2
}
```

## 6.2 実行中のモジュール操作

次の例は、輪郭抽出した動画を表示するものに変更するスクリプトである。モジュールの間に輪郭抽出フィルタを挿入し、その結果を画面に出力するモジュールに接続するように変更したものである。

**モジュールの挿入** 新しいモジュールをリンクの間に割り込ませるには、次のようなコマンドを実行する。この例では、 video ポートで結ばれている .m1 と .f1 の間に .f3 を挿入している。

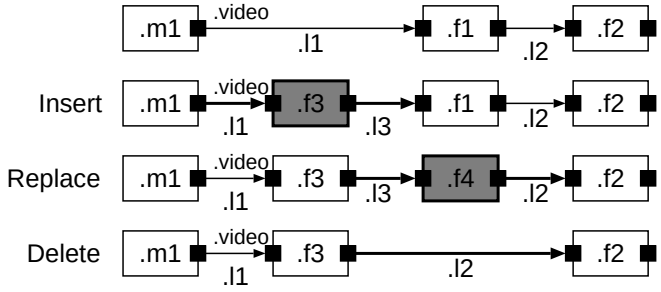



図 6.1: Module Operation

```
module .f3 -type filter3
media link .l2 connect -to .f3.video
media link .l3 connect -from .f3.video -to .f1.video
module .display -type Xdisplay
```

**モジュールの置換** 内部のコンポーネントの一部のモジュールを置換する。

この例では、video ポートを結んでいる `.f1` を`.f4` に変更する。

```
module .f4 -type filter4
media link .l3 connect -to .f4.video
media link .l2 connect -from .f4.video
module .f1 destroy
```

**モジュールの削除** モジュールを削除する。

Video ポートで結ばれている `.f4` を削除するにともなって、リンクの張り替えと余ったリンクの削除の削除を行なっている。

```
module .f4 destroy
media link .l2 connect -from .f3.video
modia link .l3 destroy
```

## 6.3 アプリケーション例

モジュール・イベント・プログラミングに基づくアプリケーションの作成を次に示す。連続メディアを処理するモジュールは、連続メディアツールキット CMT2 を用いて、C 言語で実装される。このアプリケーションは、画像を入力する `camera`、画面に表示する `xdisplay`、制御する `control` の三つのモジュールから構成されている。

```
module -newtype Control {
  event port .start -control
  event port .stop  -control
  button .b1 -text start -command {event port .start publish}
  button .b2 -text stop  -command {event port .stop  publish}
  pack .b1 .b2
```

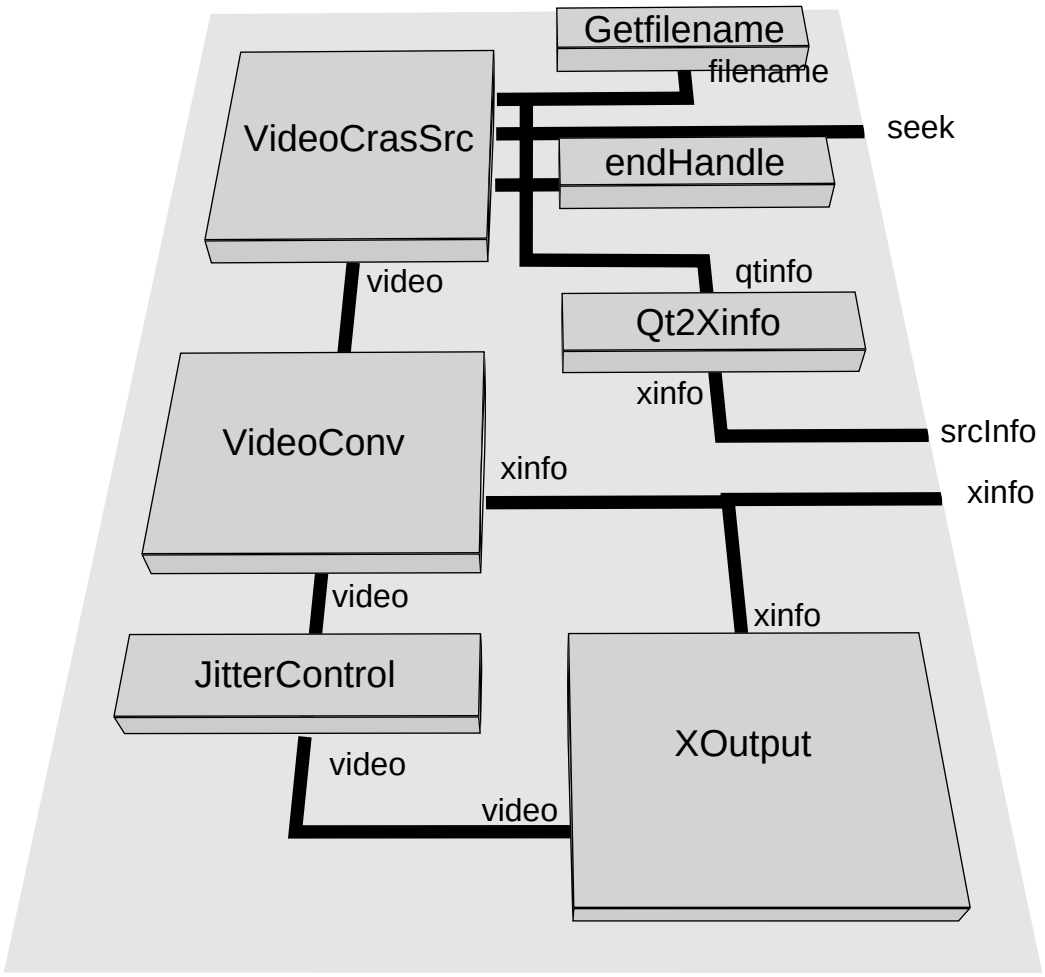

Getfilename
filename
VideoCrasSrc
seek
endHandle
video
qtinfo
Qt2Xinfo
xinfo
VideoConv
srcInfo
xinfo
xinfo
video
xinfo
JitterControl
XOutput
video
video

図 6.2: QtCmt

```
}

module -newtype Main {
  module .src -type VideoCrasSrc
  module .q2x -type Qt2Xinfo
  module .vc  -type VideoConv
  module .md  -type MovingDetect
  modlue .xo  -type XOutput
  event link .el1
  event link .el2
  event link .el1 connect -from .src.qtinfo -to .q2x.qtinfo
  event link .el2 connect -from .q2x.xinfo -to .md.xinfo -to .xo.xinfo
  media link .ml1
  media link .ml2
  media link .ml3
  media link .ml4
  media link .ml1 connect -from .src.video -to .vc.video
  media link .ml2 connect -from .vc.video -to .md.video
  media link .ml3 connect -from .md.video -to .xo.video
  pack .xo
}

event link .start
event link .stop
module .gui  -type control
module .main -type Main
event link .start connect -from .gui.start -to .main.start
event link .stop connect  -from .gui.stop -to .main.stop
pack .gui .main
```

# 第 7 章

# 考察と今後の課題

## 7.1 開発環境の特徴

本研究は、従来の連続メディアアプリケーション開発環境のモジュールの可搬性の低さに注目し、これを解決するために、二つの手法を利用した。

- out-of-band コード部のモジュール化。
- モジュールの階層化と内部操作のためのインタフェースの付加。

VuSystem の提案のように連続メディアアプリケーションを、in-band コード部と out-of-band コード部に分離してしまうのは、概念的には有効であるが、開発段階から明示的に分離してしまうのはアプリケーション・ビルダにとってはありがたいことではない。というのも、例えば、カメラの画像を流す in-band モジュールと送信画像の大きさを入力する out-of-band コードが互いに密接に関係があるように、両者はある意味で近い関係にもあるからである。

また、in-band コード部がモジュール化され交換の対象となるのであれば、out-of-band コード部にそれができないといのは、GUI に関する制限をユーザに与えることにもなる。

このように、本研究の最初の課題である out-of-band コード部のモジュール化は、従来、軽視されていた部分を見直して、対照性を取り戻したと言うこともできるだろう。

このために、構築に柔軟性が出てきたため VuSystem が提示しているような、構造以外にもより複雑な形態をとることができるようになったが、一般にどのような構成が連続メディアアプリケーションの構成にとって有用であるかは、今後の研究の課題である。

**モジュールの拡張が可能** モジュールはきわめて可搬性が高いため、あるモジュールの内部に動的にモジュールを挿入したり、置換するといったモジュールの操作が可能でありそれによって、機能の拡張を計ることができる。

また、モジュールは、連続メディアの処理だけでなく、様々なデバイスなどにも対応できる。また、イベントシステムを用いることで、さらに環境に適応するスクリプトを作成し、アプリケーションを開発することができる。

**有効な複合モジュールの提案** 複合モジュールのうち、汎用性が高いモジュールが出てくると思われる。そのような複合モジュールの定義は、毎回定義し直すのではなく、ライブラリとして利用できる良うするべきだろう。

## 7.2 イベントに関する問題

本研究では、拡張可能なモジュールを実現するためにアプリケーションの制御機構としてイベントベースシステムを採用した。

今回実装したイベントベースシステムは、非常にシンプルなものであったが、当初開発の目標としていた Qtplay[7] のようなビデオ・オン・デマンド・システム程度の規模のアプリケーションの開発環境としては十分有用であった。

しかし、in-band モジュール数が数百を越えるような巨大で複雑なアプリケーションの開発環境としてそのまま利用することは困難であると思われる。その場合、次のようなことが問題になると考えられる。

- 複雑なモジュール間の処理の優先順位をつけることができない。
- プログラマがイベントに関して記述するコードの量が膨大に、あるいは複雑になる可能性がある。

**イベントの配送** 本研究では、各モジュールのイベントの処理について、それほどシビアなタイミングが要求されないという前提と、イベントを取り込んでしまうような特殊なモジュールは存在せず、スコープ内のすべてのモジュールに配送可能であるという仕様の元に、イベントの配送順番については、単純に制御が必要なものはオーダーイベント (ordered event)、必要ないものはアンオーダーイベント (unordered event) として指定するようにした。

しかし、各モジュール間の処理に優先順位を割り付けなければ、アプリケーションが正常に機能しないという場合も考えられ、プログラマは、今回実装したより粒度の細かいイベントの配送順番の指定が必要になることがあるかもしれない。

イベントの配送の指定法には、何らかのアルゴリズムを発見して自動化するか、あるいは、プログラマによるプログラムを可能にするか、あるいはその両方と言う方法が考えられる。

プログラマブルにするという方法は、プログラマがその都度、優先順位を指定することになるが、一般的に、プログラマがこのような指定をすることなしにアプリケーションを開発できることが望ましいので、デフォルトの配送順番を持たせて、必要のない場合は指定を省略できるようにしたり、あるいはいくつかの選択肢から選ばせることで詳細に指定せずとも利用できるようにすることでプログラミングの負担を軽減することできるだろう。

**無数のイベント** 本研究で実装したスクリプト言語は、モジュールのレベルではアプリケーションの制御をイベントのみで行ない、それ以外の制御手段を提供していない。

そのためプログラマは、プログラミングに当たって二つのこと、モジュールのメディアリンクの接続、そして、モジュール間のイベントの伝達について記述すれば、アプリケーションを開発できることになる。

現在のシステムをそのまま巨大なアプリケーションに適応した場合でも、複合モジュールがうまく定義されていれば、多くのイベントを複合モジュール内に収めることができ、プログラマが扱うイベントの量はそれほど多くならないと考えている。

しかし、連続メディアアプリケーションの複合モジュールは、in-band 機能の抽象化を主目的にしたもので、場合によってはイベントポートをうまく隠蔽できず、プログラマにより大量のイベントリンクの接続が必要になる場合も考えられる。

例えば、Medusa で運用されているような中規模連続メディアアプリケーションになると in-band のモジュール数が百を超えることは少なく、それらがそれぞれが一つづつイベントリンクの接続が必要になる場合、プログラマにとって大きな負担を強いることになる。

このような問題の解決策として、イベントがリンクのつながりをつたってプロバーゲートする方法が挙げられる。これは、in-band モジュールのリンクをもとに、イベントを配送する順番および配送先を決定するものである。モジュールが関わっているストリームの送り手の方に制御を暗黙のうちに伝達し、ユーザがこの制御を記述する必要はないという特徴がある。

しかし、プロバーゲートする方法だけでは、解決し得ないことがある。例えば制御が複雑な場合、例えば選択肢が複数ある場合は、全体で一つの意志を決定することできない。また画面の大きさなどは、モジュールに依存した情報ではないので、モジュールがプロパゲートする方法では解決できない。

いずれにせよ、内部で発生するだろう無数のイベントをプログラマから有効に隠蔽する、あるい別の機構に代行させる何らかの手段が必要であろう。

# 第 8 章

# まとめ

本研究では、開発が困難な連続メディアアプリケーションの開発に関して、アプリケーション・ビルダの開発の負担を軽減する方法として、イベントシステムを導入することでモジュール化を改善することを提案した。

これにより、モジュールは、より高い可搬性を持つことができ、従来できなかったモジュール構成、操作を可能とした。

また、モジュール内部を操作するための手法も合わせて提案することで、モジュールの拡張性を維持し、拡張可能なモジュールを作成することを可能とするとし、両手法を合わせて、柔軟なアプリケーションの開発が可能な環境を提供することができた。

# 謝辞

本研究を遂行するに当たり、様々な面で支援してくださった中島研究室の皆さんと OB の方々に感謝します。また、長年に渡って御指導、御鞭撻を頂いた中島達夫教授に深く感謝します。

# 付録 A

# 定義済みモジュール

## A.1 VideoCrasSrc

QuickTime 形式のファイルを CRAS を経由して取得し、メディアストリームとして出力する。CMT の機能のうち、SetClock, SetThread は RTM が内部で暗黙のうちに処理する。ユーザが指定できるイベントポートは、qtvideo (入力)、buffer(入力) との三つである。

```
module .src -type VideoCrasSrc
```

により、ソースモジュールを生成する。ただし、これは qtcmt の下である必要がある。

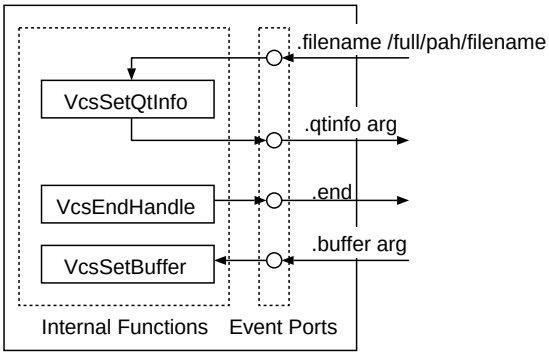



図 A.1: VideoCrasSrc Module

## A.2 Qt2Xinfo

QtVideo 形式のデータを x_info 形式に変換するモジュール。入力イベントポートとして QtVideo と出力イベントポートとして Xinfo イベントポートがある。

| |
|---|
| `.buffer` *size*(Event Input port)<br>バッファの大きさを設定する。引数としてフレーム数を指定し、例えば、毎秒 30 フレームのムービーの場合に、サイズに 30 を指定すると、一秒分のバッファが確保される。 再生中、及び、再生を一時中止した後に、これらのイベントが有効かどうかは、不明。また、再定義が可能かどうかも不明。 |
| `.qtinfo` *arg* (Event Output port)<br>QtInfo の出力通知。これは、ファイル名のイベントが到達し、それが正しい QuickTime 形式ファイルであると内部で CmtModuleAttribute がせっとされると、イベントが自動的に発送されるというもの。 |
| `.end`(Event Output port)<br>終了通知。CMT の SetEndCallback により VcsEndHandle が実行され終了イベント通知が発行される。 |
| `.filename` */full/path/filename* (Event Input port)<br>引数として QuickTime 形式のファイルを指定する。 |
| `.video` (Media Output port)<br>画像出力。CMT ではモジュールの一番の出力となっている。 |

表 A.1: VideoCrasSrc Input/Output

| |
|---|
| `.qtinfo` *arg*(Event Input port)<br>暗黙の引数として QtInfo 構造体を取得する。 |
| `.xinfo` *arg*(Event Output port)<br>暗黙の引数として x_info 構造体を出力する。 |

表 A.2: Qt2Xinfo Input/Output

## A.3 VideoConv

メディアの形式を表示可能なものに変換する。関連ソースファイル`rtmVconv.c`。

| |
|---|
| `.buffer` *size*(Event Input port)<br>バッファ設定バッファの設定する。引数としてフレーム数を指定し、例えば、毎秒 30 フレームのムービーの場合に、サイズに 30 を指定すると、一秒分のバッファが確保される。 |
| `.xinfo` (Event Input port)<br>XInfo(OutputInfo) の設定。引数は明示的に指定しない。 |
| `.video` (Media Input port)<br>画像入力。CMT ではモジュールの一番の入力となっている。 |
| `.video` (Media Output port)<br>画像出力。CMT ではモジュールの一番の出力となっている。 |

表 A.3: VideoConv Input/Output

## A.4 JitterControl

メディアのジッタを制御する。

| |
|---|
| `.buffer` *size*(Event Input port)<br>バッファ設定バッファの設定する。引数としてフレーム数を指定し、例えば、毎秒 30 フレームのムービーの場合に、サイズに 30 を指定すると、一秒分のバッファが確保される。 |
| `.video` (Media Input port)<br>画像入力。CMT ではモジュールの一番の入力となっている。 |
| `.video` (Media Output port)<br>画像出力。CMT ではモジュールの一番の出力となっている。 |

表 A.4: JitterControl Input/Output

## A.5 MovingDetec

動体検出のためのモジュール。

| |
|---|
| `.threashold` *threashold*(Event Input port)<br>閾値を設定する。 |
| `.xinfo` *density* (Event Input port)<br>センサ密度を明示的に引数として渡す。XInfo(OutputInfo) が暗黙のうちに渡される。 |
| `.detect` *points base_points* (Event Output port)<br>動体検出通知。引数として感応したセンサ数と総センサ数を持つ。 |
| `.video` (Media Input port)<br>画像入力。CMT ではモジュールの一番の入力となっている。 |
| `.video` (Media Output port)<br>画像出力。CMT ではモジュールの一番の出力となっている。 |

表 A.5: Movingdetect Input/Output

## A.6 OverRun

オーバーラン通知のためのモジュール。入出力イベントはともになし。

## A.7 XOutput

画像を出力するモジュール。関連ファイルは、`rtmXOutput.c`。

| |
|---|
| `.xinfo` (Event Input port)<br>XInfo(OutputInfo) の設定。引数は明示的に指定しない。 |
| `.video` (Media Input port)<br>画像入力。CMT ではモジュールの一番の入力となっている。 |

表 A.6: XOutput Input/Output

## A.8 複合モジュール qtcmt

ファイルから画像を読み込み、表示する。
clock、overrun、thread で指定する mach_port は内部で行なう。
**Event Ports**

| controls | |
|---|---|
| **.Start** | 再生開始 |
| **.Stop** | 停止 |
| **.Seek** | シーク |
| **.Offset** | オフセット |
| **.Qos** | QOS 設定 |
| **.OneShot** | ワンショット |

| Event Ports | | |
|---|---|---|
| **.filename** | *in* | 引数として QuickTime 形式のファイル名をフルパスで指定する。 |
| **.start** | *in* | ストリームのスタート。 |
| **.stop** | *in* | ストリームの停止。 |
| **.seek** | *in* | arg:*sec* シーク。引数で指定された時間まで映像を読みとばす。単位は秒。 |
| **.offset** | *in* | オフセットの設定。 |
| **.qos** | *in* | QOS の設定。 |
| **.oneshot** | *in* | 画像を一画面分の出力する。 |
| **.quit** | *in* | 終了 |

図 A.2: Qtplay Module

| |
|---|
| **.filename** */full/path/filename* (Input)<br>引数として QuickTime 形式のファイル名をフルパスで指定する。 |
| **.start** (Input)<br>ストリームのスタート。 |
| **.stop** (Input)<br>ストリームの停止。 |
| **.seek** *sec* (Input)<br>シーク。引数で指定された時間まで映像を読みとばす。単位は秒。 |
| **.offset** (Input)<br>オフセットの設定。 |
| **.qos** (Input)<br>QOS の設定。 |
| **.oneshot** (Input)<br>画像を一画面分の出力する。 |
| **.quit** (Input)<br>終了 |

```
module -newtype qtcmt {
  event port .filename.in
  event port .start.in
  event port .stop.in
  event port .oneshot.in
  module .src -type VideoSrasSrc
  module .cnv -type VideoConv
  module .jtc -type JitterControl
  module .mvd -type MovingDetect
  module .out -type XOutput
  module .ovr -type OverRun
  media link .l1 -from .src.video -to .cnv.video
  media link .l2 -from .cnv.video -to .jtc.video
  media link .l3 -from .jtc.video -to .mvd.video
  media link .l4 -from .mvd.video -to .out.video
  control Start   -port .start.in
```

```
    control Stop    -port .stop.in
    control OneShot -port .oneshot.in
    control Quit    -port .quit.in
}
```

**Qtplay**　Qtplay は、分散ビデオプレーヤ Qtplay を単一の in-band モジュールとして実装したものである。このモジュールの作成には、次の目的がある。

1. モジュールが複数に分割された場合とそうでない場合の、パフォーマンスの違いについて検証する
2. リンクやイベントを用いないモジュールコマンドの実装の実験を行なう。

# 参考文献


[1] Apple Computer, Inc., 1 Infinite Loop Cupertino, CA 95014 USA. *QuickTime File Format Specification, May 1996*, May 1996. URL `http://www.QuickTimeFAQ.org/the-faq/`.

[2] Shintaro Furuno and Tatsuo Nakajima. A toolkit for building continuous media applications using a new dynamic qos control scheme. Master's thesis, Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-12, JAPAN, 1995.

[3] Andy Hopper. The medusa applications environment. Technical Report 94.12, Olivetti Research Limited, 24a Trumpington Street, Cambridge CB2 1QA, England, December 1994. URL `http://www.cam-orl.co.uk/abstracts.html#49`.

[4] Shannon Jaeger. Mega-widgets in tcl/tk: Evaluation and analysis. In *Proceedings of the Tcl/Tk Workshop*, pp. 43–52, Toronto, Ontario, Canada, July 1995. USENIX Association. ISBN 1-889446-72-3.

[5] Christopher J. Lindblad and David L. Tennenhouse. The vusystem: A programming system for compute-intensive multimedia. *IEEE Journal of Selected Areas in Communications*, July 1996.

[6] Michael J. McLennan. The new [incr tcl]: Objects, mega-widgets,namespaces and more. In *Proceedings of the Tcl/Tk Workshop*, pp. 151–159, Toronto, Ontario, Canada, July 1995. USENIX Association. ISBN 1-889446-72-3.

[7] Tatsuo Nakajima and Hiroshi Tezuka. A continuous media application supporting dynamic qos control on real-time mach. In *Proceedings of the ACM Multimedia*, 1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-12, JAPAN, July 1994. URL `http://mmmc.jaist.ac.jp:8000/publications/1994/PostScript/multimedia94.ps.gz`.

[8] John K. Ousterhout. *Tcl and the Tk Toolkit.* University of Calfornia Berkele, December 1993.

[9] Progressive Networks, Inc., 1111 3rd Ave, Suite 2900 Seattle, WA 98101 USA. *RealVideo Technical White Paper*, February 1996. URL `http://www.real.com/products/realvideo/`.

[10] Progressive Networks, Inc., 1111 3rd Ave, Suite 2900 Seattle, WA 98101 USA. *White Paper on RealAudio Client-Server Architecture*, 1996. URL `http://www.realaudio.com/prognet/openarch/`.

[11] Frank Stajano. Writing tcl programs in the medusa applications environment. Technical Report 94.7, Olivetti Research Limited, 24a Trumpington Street, Cambridge CB2 1QA, England, July 1994. URL `http://www.cam-orl.co.uk/abstracts.html#30`.

[12] Frank Stajano and Rob Walker. Taming the complexity of distributed multimedia applications. In *Proceedings of the Tcl/Tk Workshop*, pp. 213–228, Toronto, Ontario, Canada, July 1995. USENIX Association. ISBN 1-889446-72-3. URL `ftp://ftp.orl.co.uk/pub/docs/ORL/tr.95.5.ps.Z`.

[13] Hiroshi Tezuka and Tatsuo Nakajima. Design and implementation of a continuous media storage system on real-time mach. Research report, Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-12, JAPAN, July 6 1994.

[14] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time mach: Towards a predictable real-time system. In *Proceedings of Usenix First Mach Symposium*, pp. 73–82, Burlington, Vermont, October 1990. USENIX Association. URL `http://mmmc.jaist.ac.jp:8000/publications/1990/PostScript/usenix90.ps.gz`.

[15] Mark L. Ulferts. [incr widgets] an object oriented mega-widget set. In *Proceedings of the Tcl/Tk Workshop*, pp. 61–76, Toronto, Ontario, Canada, July 1995. USENIX Association. ISBN 1-889446-72-3.

[16] David Wetherall and Christopher J. Lindblad. Extending tcl for dynamic object-oriented programming. In *Proceedings of the Tcl/Tk Workshop*, pp. 173–181, Toronto, Ontario, Canada, July 1995. USENIX Association. ISBN 1-889446-72-3. URL `ftp://ftp.orl.co.uk/pub/docs/ORL/tr.95.5.ps.Z`.

[17] 大平浩貴. 連続メディアデータ処理とそのアプリケーションの製作を支援するツールキットに関する研究. 修士論文, 北陸先端科学技術大学院大学 情報科学研究科, 923-12 石川県能美郡辰口町旭台 1-1, 1997.