VDM-RT for Co-simulation

John Fitzgerald
Peter Gorm Larsen

AARHUS
UNIVERSITY

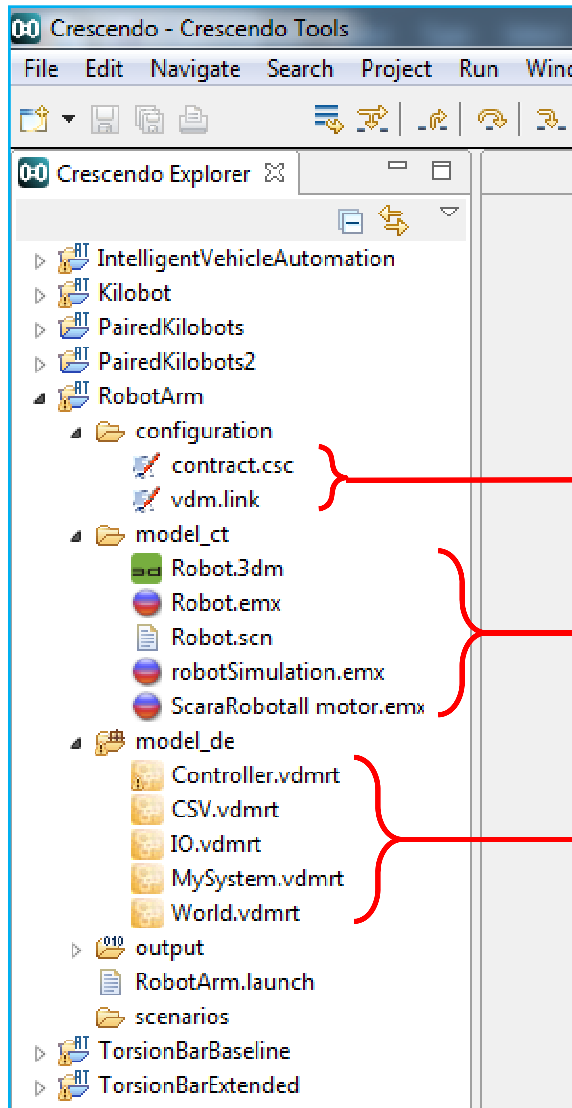
背景: VDM

- ゴール: 根拠に基づきつつ利用しやすいモデリング・分析の技術
- VDMTools → Overture → Crescendo → Symphony
  - 実用的な開発方法論
  - 産業応用
- VDM: モデル規範型の仕様記述言語
  - オブジェクト指向と実時間性による拡張
  - 静的解説のための基本ツール
  - シミュレーションの強力な支援
  - モデルベーステスト

概要

- CrescendoでのVDMの活用
- VDM-RT (Real-Time)
  - クラス，インスタンス変数，関数，操作，定数，スレッド，同期
  - 実時間性の扱い
- CrescendoにおけるDE-firstモデリング
  - 近似のモデリング

Crescendo Workspace



- 名前空間（パッケージ）はない
- *model_de* ディレクトリ以下のVDM-RTクラスがすべてチェックされる
- 自動補完は現時点でなし・・・

デバッギング

- 出力

```
IO`print(“a string”)
IO`println(“a string plus newline”)
IO`printf(“%s: value of x is %s”, [1, x])
```

  - 現在は %s のみサポートis supported currently!
  - 文字列連結の演算子は ^ (通常Shift-6)
  - IOの後の ` (バッククオート)はクラスのstatic要素へのアクセスに用いる (Javaと異なりstatic要素には異なる文字を用いる)
- ブレイクポイントの設定・Debug perspectiveの利用

単純なControllerクラス

```
class Controller

instance variables

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := P(err);
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

Co-simulation エンジンはこれらの変数を20-simモデルと同期

- instance variables, operationsなどキーワードでブロックを区切って記述(同じ種類の記述が任意の順序で何度出てきてもよい)
- 継承

```
class Controller is subclass of Parent
```

- オブジェクトの生成

```
new Controller
```

- コンストラクター定義はJavaと同様

```
public Controller: real * real ==>
Controller
Controller(a,b) == (
    x:= a;
    y := b
);
```

- コメントは
  - ハイフン2つから行末 `-- comment`
  - または `/* block comment */`

Instance Variables（インスタンス変数）

```
class Controller

instance variables

private measured: real := 0;
public setpoint: real := 0;
protected err: real := 0;
output: real := 0;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := P(err);
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- オブジェクトの状態を保持
- 型を与え宣言する文法はJavaなどと違う

  `private measured: real;`

- アクセス修飾子はJavaと同様（個々では説明のためにいろいろ付けてみている）
  - 省略した場合 **private**
- 定義時に初期値を設定できる
- 型については後述（**real**）など

Functions（関数）

```
class Controller

instance variables

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := P(err);
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- 関数はpure
  - 副作用なし
  - インスタンス変数にアクセスしない
- returnキーワードは使わない
  - 戻り値を与える式自体が関数の定義となる
- 補助計算の定義に便利
- 関数定義の初めにシグネチャーを書く

  `real * int * bool -> real`
- ループは使わない
  - 関数型プログラミング
  - 再帰呼び出しなど関数呼び出し

Operations（操作）

```
class Controller

instance variables

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := P(err);
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

void相当

- 関数と同様だが
  - インスタンス変数にアクセス可能・副作用を持てる
  - Javaなど同様に手続き的記述
  - While文によるループなど記述可能
  - return文により戻り値を与える
- 関数または操作をさらに呼び出せる
- ローカル変数を定義できる（ただし操作の開始時点に限る）

```
Step() == (
     dcl x: real := 0;
```

- 複数の文の順次実行は（　）で囲む（Javaなどでの｛　｝を使わないように）
- 関数とはシグネチャー定義の矢印が異なる

```
real * int * bool ==> real
```

Values（定数）

```
class Controller

instance variables

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := P(err);
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- 値の指定は代入文の := ではなく = を使う
- 型宣言は必須ではないが指定してもよい

```
Kp: real = 1.24;
```

- Static要素扱いとなる

```
Controller`Kp
```

Threads（スレッド）

```
class Controller

instance variables

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := P(err);
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- そのクラスのオブジェクトにスレッドを割り当てて実行する処理を定義
- その定義は1回の操作呼び出し

```
thread
Step();
```

- またはループ

```
thread
while true do Step();
```

- スレッドの割り当て・開始

```
ctrl: Controller := new Controller();
start(ctrl)
```

- または，左の例のように定期的な実行を定義
  - 2e7ナノ秒ごとに Step 操作を呼び出し（20ミリセカンド，0.02秒，50Hz）

VDM-RTの重要な特徴 (1)

- VDM-RT (Real Time) はリアルタイムシステムのための拡張を含む
- 内部クロック
  - シミュレーション開始時からのナノ秒表現
  - **time** キーワードによりアクセス可能
    - **dcl** now: **real** := **time**/1e9 -- time in seconds
- 「すべての」式はクロックを進める
  - デフォルトでは2クロックサイクル
  - 明示的に変更することもできる

```
cycles(number)(expression)
duration(number)(expression)
```

VDM-RTの重要な特徴 (2)

- 内部クロックは20-simと同期される（前述の意味論の通り）
- CPUおよびバスのモデルにより，実際のコード実行をモデリング
  - オブジェクトは，速度が設定されたCPUに割り当てられる
  - CPU速度に応じて実行時間が変わる
  - クロックを進めない仮想的なCPUのモデルも（オブジェクトが割り当てられていない場合）

Systemクラス

```
system MySystem

instance variables

-- controller
public static ctrl: Controller;

-- CPU
private cpu: CPU; := new CPU(<FP>, 1E6)

operations

public MySystem: () ==> MySystem
MySystem() == (
    ctrl := new Controller();
    cpu.deploy(ctrl)
)

end MySystem
```

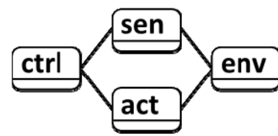
- CPUおよびその割り当てのための特別なクラス
- インスタンス変数とコンストラクタの未定義
- （シミュレートされた）MIPSによるCPU速度の表現
  - 実際のものの20%以下のモデルが典型的には「十分よい」

Worldクラス

```
class World

operations

-- run a simulation
public run: () ==> ()
run() == (
    start(System`ctrl);
    block();
);

-- wait for simulation to finish
block: () ==> ()
block() == skip;
sync per block => false;

end World
```

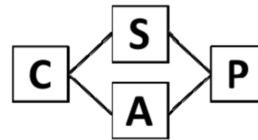
- コード実行の入り口
- run()操作がJavaなどのmain()に相当
- スレッドを開始しシミュレーションの終了を待つ

DE-firstモデリング (1)

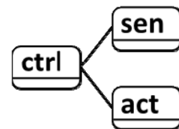


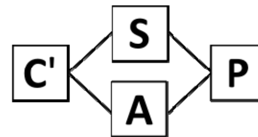
DE-first development

Contract definition



CT-only modelling





Integration of initial co-model

- DE-first (DE-only) モデル :
  - コントローラー，センサー，アクチュエーターのクラス
  - *環境のモデル*

DE-firstモデリング (2)

- DE形式化に基づくシステムモデルから開発を開始
- このモデルには，コントローラー（ctrl）と環境（env）のオブジェクトを含む
- センサーとアクチュエーターオブジェクトにより連結（sens・act）
- 環境オブジェクトにより，CTの世界をDEドメインにて模倣する
- 十分な確信が得られたら契約を定義
- センサーとアクチュエーターのオブジェクトの別の実装を用意
  - 環境オブジェクトではなく，co-simulationエンジンにより更新される共有変数の置き場として振る舞うようなものに

環境モデル

- 後にCTモデルにより置き換えられるプラントの単純化されたモデル
- スレッドとして振る舞い（あるいはスレッドから呼ばれる）`Environment` クラスを作成する
  - 操作を `dt` 時間ごとに
- 2種類のアプローチ
  - データ駆動：事前に計算されたデータを読み込み，センサーオブジェクトを通してコントローラーに提供
  - 統合：CTのような統合器を実装する
  - またはこれらの組み合わせ

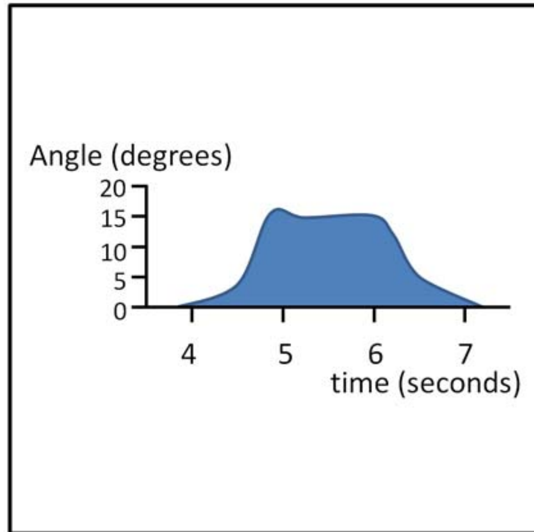
単純な統合

- 時間ステップ dt でシミュレーションされる，加速度，速度，位置が定義された移動オブジェクトを考えてみる
- 単純なオイラー統合：

```
position = position + velocity * dt;
velocity = velocity + acceleration * dt;
```

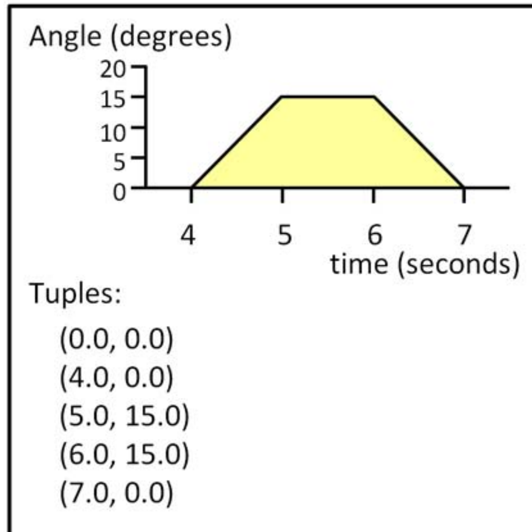
- 仮定が単純化されている
  - 加速度が定数，
  - モーターには加速がなく瞬間的に速度が得られる，など

CT振る舞いの近似

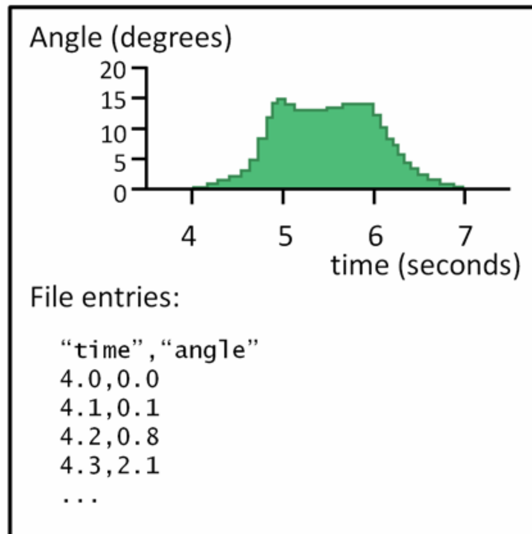


- プラントモデルに対して線形近似は使えるが，非線形な場合は？（例：ユーザー入力）
- 例：self-balancing scooterに対するユーザー入力のプロット
  - 極めて忠実
  - 安全性や，起動時などのモードのテストのためには，近似のみでも

Finding Approximations



- タプル
  - 時間･値のペアの列を作成する
  - seq of (real * real)
  - 指定時間に変化することとし，それらの間を補完する



- データ入力
  - 測定値または生成されたデータを用いる
  - CSVファイルに記録し指定時間に読み込む

```
CSV`freadval[seq of real](filename)
```

まとめ

- CrescendoにおけるコントローラーのためにVDM-RTを用いる
  - 継承のサポートなどのオブジェクト指向
  - クラスの定義はブロックに分けられる
    - instance variables, operations, functions, values, thread, sync
  - 20-simと同期される内部クロックがあり，すべての式は時間を経過させこの内部クロックを増加させる
- DE-first
  - 単純化されたプラントのモデル
  - 単純なシミュレーターのようにスレッドとして走る
  - CT振る舞いの近似

演習：
Line-following Robot Co-model

John Fitzgerald
Peter Gorm Larsen

AARHUS
UNIVERSITY

進め方

- *Practical.zip* を解凍し，*Practical-Instructions.pdf* を参照